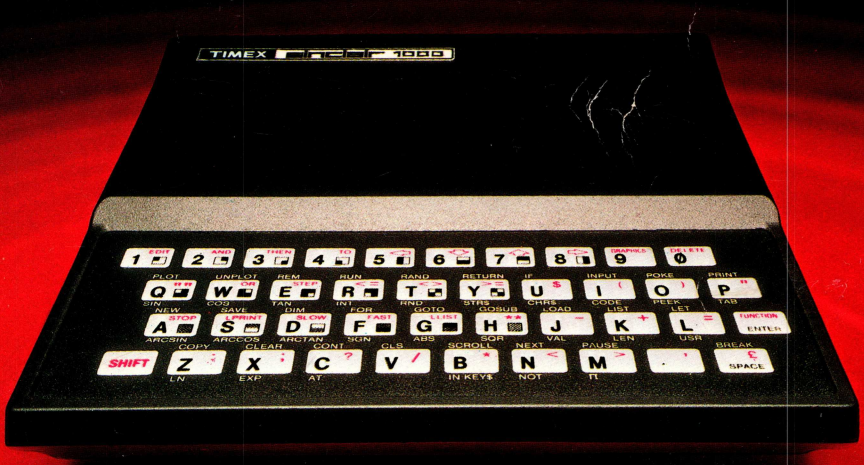
TIMEX

User Manual



TIMEX sinclair 1000

TIMEX User Manual

by Steven Vickers
with revisions by
C. F. Durang

© 1982 by Timex Corporation
© 1982 by Sinclair Research Limited

TIMEX sinclair 1000

This equipment generates and uses radio frequency energy and if not installed and used properly, that is, in strict accordance with the manufacturer's instructions, may cause interference to radio and television reception. It has been type tested and found to comply with the limits for a Class B computing device in accordance with the specifications in Subpart J of Part 15 of FCC Rules, which are designed to provide reasonable protection against such interference in a residential installation. However, there is no guarantee that interference will not occur in a particular installation. If this equipment does cause interference to radio or television reception, which can be determined by turning the equipment off and on, the user is encouraged to try to correct the interference by one or more of the following measures:

- reorient the receiving antenna
- relocate the computer with respect to the receiver
- move the computer away from the receiver
- plug the computer into a different outlet so that computer and receiver are on different branch circuits.

If necessary, the user should consult the dealer or an experienced radio/television technician for additional suggestions. The user may find the following booklet prepared by the Federal Communications Commission helpful: "How to Identify and Resolve Radio-TV Interference Problems". This booklet is available from the US Government Printing Office, Washington, DC 20402, Stock No. 004-000-00345-4.

WARNING: This equipment has been certified to comply with the limits for a Class B computing device, pursuant to Subpart J of Part 15 of FCC Rules. Only peripherals (computer input/output devices, terminals, printers, etc.) certified to comply with the Class B limits may be attached to this computer. Operation with non-certified peripherals is likely to result in interference to radio and TV reception.

This edition revised and produced by
The Communications Company,
North Attleborough, MA.

This manual was typeset electronically
by WordPower, Burlington, MA, on a
WordPower Composition System.

Table of Contents

Chapter	*Topic*		*Page*
	Using the Computer		
	Introduction: Getting Started		1
1	**How to Set Up Your T/S 1000**		5
2	**Using Ready-to-Run Programs**		9
3	**Telling the Computer What to Do**	**PRINT, ENTER, DELETE,** ◊ , ◊	15
	Elementary Programming		
4	**Writing a Program**	**NEW, GOTO, RUN, CONT, BREAK**	21
5	**Punctuation and Arranging Output on the Screen**	Comma, Semicolon, ◊, ◊, **EDIT, TAB,** Functions	23
6	**Loops and IF**	Variables, Loops, **LET, STOP, IF, FOR, NEXT, TO, STEP, LIST, REM, INPUT, PI, CLEAR**	27
7	**More About IF**	**THEN, AND, OR, NOT,** =, <, >, <=, >=, <>	31
8	**SLOW and FAST**	**SLOW, FAST**	39
9	**Subroutines**	**GOSUB, RETURN**	41
10	**When the Computer Gets Full**		47
11	**Mathematics with the T/S 1000**	+, −, *, /, **	51

Advanced Programming

12	**Advanced Printing Techniques**	**AT, TAB, CLS, SCROLL**	55
13	**The Character Set**	**CODE, CHR$**	57
14	**Graphics**	**PLOT, UNPLOT**	63
15	**Functions**	**RAND, RND, ABS, SGN, SIN, COS, TAN, ASN, ACS, ATN, LN, EXP, PI, SQR, INT**	69
16	**Time and Motion**	**PAUSE, INKEY$**	75
17	**Arrays**	**DIM**	79
18	**Strings**	**LEN, VAL, STR$**	85
19	**Substrings**	Slicing with **TO**	89
20	**Sinclair BASIC Print Commands**	**LPRINT, LLIST, COPY**	93
21	**The T/S 1000 for Those Who Understand BASIC**		97
22	**Flowcharting and Debugging**		111
23	**Number Systems**		115
24	**How the Computer Works**		119

For Experts Only

25	**Using Machine Code**		123
26	**Organization of Storage**		127
27	**The System Variables**		133

Reference

	Appendix — The Character Set		137
	Index		144
	Report Codes		153

LIMITED WARRANTY

Basic Coverage: This Timex/Sinclair Computer is warranted to the owner for a period of 90 days from date of original purchase against defects in manufacture. This limited warranty is given by Timex Computer Corporation — not by the dealer from whom it was purchased.

What Timex Will Do: If a defect in manufacture of the Computer is discovered within 90 days from date of original purchase, Timex Computer Corporation will, at its option, repair or replace the defective unit.

What You Must Do: You must return the Computer, indicating date of purchase, to Timex Product Service Center with a written explanation of the reason for the return.

Return your unit, postage pre-paid, to:

Timex Product Service Center
P.O. Box 2740
7000 Murray Street
Little Rock, AR 72203

To protect against in-transit loss, we recommend you insure your Computer.

Limitations:

THE ABOVE REMEDY IS EXCLUSIVE. TIMEX COMPUTER CORPORATION LIMITS THE DURATION OF ANY WARRANTY IMPLIED BY STATE LAW, INCLUDING THE IMPLIED WARRANTY OF MERCHANTABILITY, TO 90 DAYS FROM THE DATE OF ORIGINAL PURCHASE. TIMEX COMPUTER CORPORATION IS NOT LIABLE FOR ANY INCIDENTAL OR CONSEQUENTIAL DAMAGE. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state. Some states do not allow limitations on how long an implied warranty lasts, or the exclusion or limitation of incidental or consequential damages, so the above limitations or exclusions may not apply to you.

This warranty is void if the Computer has been tampered with or ill-treated or if the defect is related to servicing not performed by us.

Introduction

Getting Started

Welcome to the world of computing. Before you plug in your new Timex/Sinclair 1ØØØ, please take a moment to think about this exciting new adventure. We want to assure you that:

1. You will enjoy computing.

2. You will find it easy as well as enjoyable.

3. You shouldn't be afraid of the computer. You are smarter than it is. So is your parakeet, for that matter.

4. You will make mistakes as you learn. The computer will not laugh at you.

5. Your mistakes will not do any harm to the computer. You can't break it by pushing the "wrong" button.

6. You are about to take a giant step into the future. Everyone will soon be using computers in every part of their daily lives, and you will have a head start.

You do not need to know how to program a computer to use the T/S 1ØØØ, any more than you need to know how to do a tuneup to drive a car. You may *want* to learn to program — it is not difficult and can be very enjoyable — but you can use the computer for the rest of your life without having to learn programming.

A computer is a tool, like a hammer or saw — or perhaps like a food processor. Hammers and saws generally do only one thing well. A food processor can perform different operations, and normally you can "program" it by simply pushing the proper buttons. A computer is an information tool, and is the most versatile tool ever invented. Because it can do many things, it needs a sequence of instructions to perform any particular task. These instructions are called *programs*.

There are many available programs for your everyday use with your Timex/Sinclair 1ØØØ. You can use them for learning, and for home or business management (like balance-sheet calculations, record-keeping, accounting/bookkeeping, taxes, personal or business inventories, etc.). You can maintain athletic statistics, recipes, address or Christmas-card lists, prepare reports, learn and use mathematics, play games, and do many other things.

One of the most important uses of the Timex/Sinclair computer is as an educational tool. Right now your children are beginning to use computers in school. They are learning about computers, and they are using computers to help them learn other subjects. Your T/S 1ØØØ can help your children learn at home, whether their schools have computers or not. Many educational programs are available, for both tutorial help and advanced learning. You can find all of these programs at the same store where you bought your Timex/Sinclair 1ØØØ. Many more programs are being developed right now for the T/S 1ØØØ, because it is the world's best-selling computer. In the near future, your personal computer will be able to dial and answer your telephone, monitor your burglar alarm, control appliances, water your lawn and perform many other duties for you. Keep in touch with your dealer!

About This Guide

This book is in several sections, as you can see from the Table of Contents. The first section tells you how to plug in and set up your Timex/Sinclair 1ØØØ, and use programs from books and tape cassettes. The second part is an elementary introduction to programming, so that you can, if you wish, learn the basics and better understand how the computer does its work. Later sections get into more advanced programming and provide reference material for experts.

Some of the later chapters may be especially valuable to you if you want to program for particular uses:

Chapter 21, "The T/S 1ØØØ for Those Who Understand BASIC," is an introduction to the specifics of Sinclair BASIC as used by the T/S 1ØØØ, and is a good starting place for experienced programmers — but it is also a good review for beginners when they have reached that point in the book.

If you want to try your hand at writing games with moving graphics, or make artistic designs on the screen, you'll want to see Chapter 13, "The Character Set," 14, "Graphics," and 16, "Time and Motion." Eventually, you'll even want Chapter 25, "Using Machine Code."

For the mathematically or scientifically inclined, Chapters 11, "Mathematics with the T/S 1000," and 15, "Functions" will be of most interest, and if you plan to manipulate text material, you should see Chapter 12, "Advanced Printing Techniques," 18, "Strings," and 19, "Substrings," as well as perhaps 17, "Arrays."

Have fun as you get to know your Timex/Sinclair 1000 computer!

Chapter 1

How to Set Up Your T/S 1ØØØ

We've provided everything you need to start using your T/S 1ØØØ computer immediately, with your own television set and an ordinary audio cassette recorder.

Here are the components of your personal computer system:

1. Your *television set*. You can use a color or a black-and-white set but, of course, the Timex/Sinclair 1ØØØ will display in black and white only.

2. A *transfer switch box*, which enables you to switch between your TV antenna and the computer.

3. A *cable*, about four feet long, to connect the computer to your TV.

4. The computer itself. You will see three jack sockets on one side, marked 9V DC IN, EAR and MIC, a larger socket marked TV, and, on the back, an open space in which you can see an edge of the circuit board inside.

5. The *power supply*. It is a transformer that plugs into any standard wall socket (110 volts), with a cord that plugs into the computer's 9V DC IN socket. (You will be glad to hear that you can touch the plug at the computer end without getting any shock at all, and can plug it into the wrong socket in the T/S 1000 without doing any damage.) The computer has no on/off switch; you turn it on by simply plugging it in with the power supply.

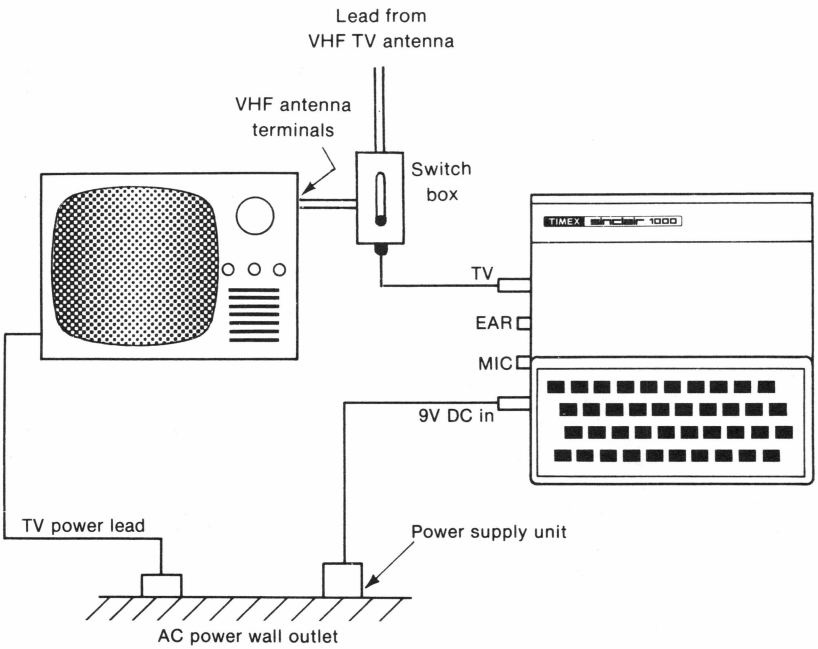
6. A *double cable*, about a foot long, with 3.5 mm plugs at each end, to connect the T/S 1000 to your tape cassette recorder.

7. Your *cassette recorder*. You'll need one that will accept the 3.5 mm plugs in its earphone and microphone sockets. (If you have a recorder and it doesn't fit the plugs, try taking the recorder and the cables to your local electronics shop. You will probably be able to get very inexpensive adaptors rather than buying another cassette recorder.) Strangely enough, any inexpensive battery-powered recorder will usually work best with the T/S 1000 and, in fact, expensive and powerful stereo equipment may damage the computer. It is helpful — but not essential — if your cassette recorder has a *counter*. By measuring how much tape has gone by the recording/playing mechanism, the numbers in the counter window can help you find programs on your cassettes.

Now that you have all the parts assembled, you can put the system together. (If you are going to try some programming on your own, you can do without the recorder and skip to Chapter 3 after setting up; if you have a recorder and want to use some pre-recorded software, or save on tape some of your own programming, you'll want to see Chapter 2.)

First, disconnect the VHF TV antenna wires from your television set (you can leave the UHF wires alone). Connect the wires from the transfer switch box to the terminal on your TV instead, and then connect the antenna wires to the screws marked TV on the transfer switch box. Plug the four foot connecting cable into the socket on the transfer switch box and into the TV socket on the T/S 1000.

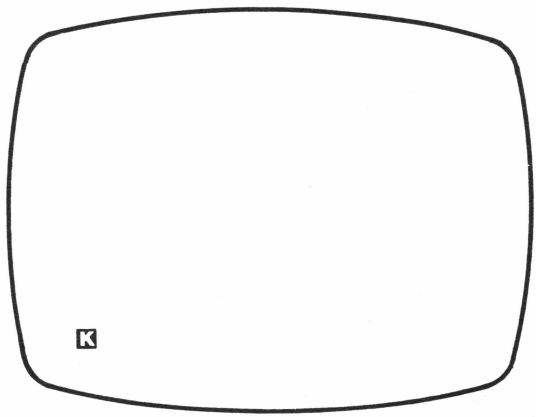
> NOTE: If you have cable TV, or a 75-ohm antenna lead (a round wire ending in a screw terminal), you will need to pick up a small device to convert this to the standard, flat, two-wire antenna lead that connects to the transfer switch box. There are several versions of this device, which may be called a "UHF/VHF matching transformer," "75-to-300-ohm converter," "cable adaptor" or "VCR adaptor." Someone at your local electronics store will be able to help you; the cost will be from three to ten dollars. You *may* have to contact your cable company if their wire goes into your set instead of screwing on to the terminal on the back.



Second, plug the power supply into the wall and into the 9V DC IN socket on the computer.

Third, turn on the TV. Set it to channel 2 or channel 3, whichever one is not being used for broadcasting in your area. Make sure the switch on the bottom of your T/S 1ØØØ is set to the same channel. Turn the sound all the way down.

You should have a picture like this on your screen:



The K in the lower left-hand corner of the screen is called the *cursor*, and means the T/S 1ØØØ is ready for action.

Fourth, connect your recorder to the computer with the twin leads. Although we've provided two, it may be best to use them one at a time (you can experiment); connect the earphone socket on the recorder to the EAR socket on the computer in order to load a program from tape into the T/S 1ØØØ, and the microphone socket of the recorder to the computer's MIC socket to save programs you've put information into or written.

NOTE: The picture on your TV screen should be clear; if you are getting interference, try the following steps in order:

1. Adjust the tuning control on the set, then try the brightness, contrast, and horizontal hold (horizontal is usually on the back of the set).
2. Move the computer away from the TV set; or, if possible, place it lower than the set.
3. Plug the computer into a different circuit from the television set; usually outlets on opposite walls are on different branch circuits.
4. You may wish to try a longer (shielded) cable between the switch box and the computer to move the T/S 1ØØØ still farther away from the TV.
5. Consult an experienced radio/TV repairman; your set may need adjusting.

Now you're ready to use your Timex/Sinclair 1ØØØ!

Chapter 2

Using Ready-to-Run Programs

As of now, there are many programs available for the Timex/Sinclair 1000 computer, programs which are fun and programs which are useful. It is exciting to see how many new ones appear each month. You will find that programs written for the Sinclair ZX81, the predecessor of the T/S 1000, will run on your computer.

Some programs — the longer ones — may require the T/S 1016 16K RAM Pack, which can be attached to the back of the computer to increase the computer's *memory* capacity; i.e., the amount of information it can hold. The 16K RAM Pack is available where you purchased your computer.

Sometimes you will load a program into the computer from a tape cassette. You can then use the program, and even re-use it as often as you like. As long as the computer remains on, it will retain that program. When you are done and turn off the computer, it will disappear from the T/S 1000's internal memory but, of course, you will still have it on tape to load and use again. This is how you will usually use games, for example.

Sometimes you will type in a program from a book in order to use it, and then will want to save it on a tape cassette. Then, the next time you want to use it, you won't have to type it in, but simply load it from the cassette.

And sometimes there will be programs you will load from a tape, add data to, and then save the program with the added data on another part of the tape, separate from the original program.

Let's look at how each of these is done.

Loading a program from tape

Every program should have a name, and any cassette that has more than one program on it should provide you with an index listing the names of all the programs on the tape. Often this index will mark the location of each program with a tape counter number. If you set the counter to ØØØ, then run the tape forward to the number of a program you want to use, you should be at that program's approximate location. (Caution: since cassette recorder counters are usually not exact, it is a good idea to stop a few numbers short of the index figure — at Ø33 if the index calls for Ø37, for instance.)

With all the components of your system connected and turned on, as discussed in Chapter 1, make sure that your tape is rewound to the beginning, and that the K cursor is on your TV screen.

Connect the EAR socket on the computer to the "earphone" or "headphones" socket on the tape recorder to about three-quarters of the maximum volume. If it has tone controls, adjust them so that treble is high and bass is low.

Then type

LOAD

which is what you get when you press the J key while the K cursor is showing. (Whenever the K cursor is on the screen, pressing any key will give you the *keyword* command printed above that key on the keyboard.) Notice that the computer has printed

LOAD

on the screen.

You'll notice also that the cursor has changed to L. This means that the computer now will interpret any key you press as the *letter* or main (large) symbol on any key. (If you pressed the J key again, for instance, you'd get a J.)

If you want one of the symbols printed in *red* on a key, you hold down the **SHIFT** key (notice that **SHIFT** is in red, too) and press the key you want. That is what we will do:

You want to instruct the computer to load the program you wish to use, so you must put the name in quotes. Suppose you want to run a program for a game, called STAR ZAP.

Hold down the **SHIFT** key and press the P key, and you'll get quotation marks. Then, without **SHIFT**, type in the name of the program, making sure you have it exactly right including spaces. Then type **SHIFT** P again for quotation marks. Your screen will look like this:

LOAD "STAR ZAP"

(If you make a mistake, you can start over by pressing **NEW**, and then **ENTER**, or by briefly unplugging the computer. In Chapter 3, we will show you how to easily make corrections without starting over.)

Now start the cassette recorder (on PLAY), and then type

ENTER

You will see patterns of black and white stripes on your TV screen — thin, squiggly ones while the computer is searching for the program you've specified, and thick straight ones while the program is being loaded into the T/S 1ØØØ. When it is ready, the black and white stripes will stop, and in the lower left-hand corner of the screen will be the "report" Ø/Ø. (This means that the computer has successfully completed the loading assignment.) You can then turn the tape recorder off.

If the entire tape plays — or so much of it that you suspect that your program has not loaded properly — and you don't get the Ø/Ø report, press the **BREAK** key. Something has gone wrong and bears a little investigation. (A short program should not take more than fifteen seconds to load, once the heavy lines start.)

For instance, it is possible that the tape was not positioned properly (remember when we told you those counters were not always exact?). The computer might not have found the beginning of the program you desired. (This is indicated if the thin, squiggly lines never changed to thick, straight ones.) Double check the program location, back up the tape to a point a bit earlier than before, and try again.

The next most likely problem is that the volume level is too high or too low. It needs to be (a) loud enough for the computer to pick up the program, (b) not so loud that the program is distorted (this is actually fairly rare), and (c) quiet enough for the silent part to be recognized as silent by the computer.

The best adjustment is to turn the volume up as loud as it will go without causing silent spaces on the tape to be noisy; you can check this by disconnecting the plug in the recorder's earphone socket and listening to the tape on the speaker. If the silence is very noisy, either you still have the volume set too high or you may have other problems:

Some tape recorders can form a feedback loop with the T/S 1ØØØ. (This means that output from the recorder gets mixed up with input to the recorder, resulting in distortion of the signal.) This is why we recommended connecting only one lead at a time; if you've recorded something with both EAR and MIC connections made, you may not be able to load it.

Some tape recorders can record a 6Ø cycle AC hum. This can be avoided by operating them on batteries.

Some tape recorders — especially old, worn ones — are intrinsically noisy, and produce a lot of extraneous noise on their tapes. You may have to invest in another recorder.

You may have to wiggle the plug in the earphone socket; on some recorders contact is lost if the plug is pushed in too far. If you pull it out just a bit, you may feel it settling into a more secure position.

It is possible to load a program without using its name. If you type

LOAD " "

then start your recorder and press **ENTER**, the T/S 1ØØØ will load the first program it comes to. You can use this method if you've forgotten the name of a program. You can even load and use a number of programs on a tape, one after another. (When you stop the recorder after you have finished loading one program, the tape will be in position to load the next.)

When you have successfully loaded a program (verified by the Ø/Ø report), you can use it by pressing

RUN and **ENTER**

and then following the instructions the program itself gives you on the screen, and any printed instructions.
(You can look at the program by typing

LIST and **ENTER**

and *then* **RUN** it, if you like.)

Typing a printed program and saving it on tape

Many shorter programs are available in books and magazines. You can use them by simply typing them in. Type them exactly as they appear in the publication, making sure your spellings are correct, and all punctuation and spaces as well.

You can check your listing by comparing what you have on the screen with the printed version. See Chapter 3 for how to easily make corrections.

When you've finished typing the program in, you execute it by pressing

RUN and **ENTER**

as above. When you've finished using it — either you reach the end, or you interrupt the program by pressing the key marked **BREAK** — you can get the listing back on the screen by pressing

ENTER

again. (You don't *have* to type **LIST** in Sinclair BASIC.) Then, after verifying (by using it) that the program works and that you've typed it in correctly, you can save it for future use on tape. (You don't have to try it out first, of course, but then you might be going to the trouble of saving a program that won't run.)

Saving a program on tape

As we said earlier, every program should have a name. The T/S 1ØØØ, in fact, won't save a program on tape without a name. You can make up a name for a program you invent, use the name of a program you have typed in as above, or even change that name to something you like better. Whatever you

call the program when you save it will be the name you have to ask for to load it later.

> NOTE: It is a good idea to put the name of a program into the listing of that program, so you can doublecheck that you have the right one. The easiest way is to use a REM line at the beginning. **REM** — the *keyword* over the E key — means *remark* or *reminder*. Any line in a program that starts with **REM** is not used by the computer, but is shown on the screen to people to help them use the program.
>
> The Timex/Sinclair 1ØØØ automatically puts the lowest-numbered line of a program first, and arranges all the lines in numerical order. You can put a new first line — a **REM** statement — in a program, after you have loaded it by typing
>
> 5 **REM** "STAR ZAP" **ENTER**
>
> using a line number lower than the lowest in the listing (you'd use 5 if, for instance, the first line of the program started with the number 1Ø) and, of course, the actual name of your program.

Connect the MIC socket of the computer to the microphone socket of the recorder. Position the tape in a part that is blank, or a part that you are prepared to overwrite. Type:

SAVE "STAR ZAP"

Start the tape recorder, on RECORD, then press

ENTER

Watch the TV screen. You'll see the pattern of black and white lines, and eventually the pattern will end and the screen will show Ø/Ø.

As a check on whether the recorder has received the program correctly, you can listen to it over the speaker. Rewind the tape to where you started and play it back. You should hear, in order

1. A soft, humming buzz. This is the signal before you pressed **ENTER**.

2. Five seconds of silence.

3. A very harsh, high-pitched buzz, which is the program itself. This will go on longer for a longer program.

4. The soft, humming buzz again.

NOTE: A technique that may make it easier to find programs you record is to speak the name of the program onto the tape through the microphone before connecting the computer to the recorder. Then you can search for the sound of your announcement to find the program.

You can check on your success at saving a program by following the **LOAD** instructions above.

Saving programs with your own data entered

Some programs are meant for you to enter your own data into — saving lists, figures, etc. These are easily used by following the same procedures we've just discussed.

1. **LOAD** the program as we've described.

2. **RUN** the program, entering your own data as it is called for.

3. **SAVE** the program with data in it, using a new name to distinguish it from the original program. If, for example, you load a program called "Calculator" and then fill in your personal financial records, you may want to save the filled-in version under the name "Finances."

As you can see, there are many ways to use your Timex/Sinclair 1000 without learning computer programming. But if you'd like to look into it, try the next chapters and see how you like it.

Chapter 3

Telling the Computer What To Do

Now that you have the screen looking like the picture in Chapter 1, you can enter special computer "instructions." For example:

PRINT 2+2

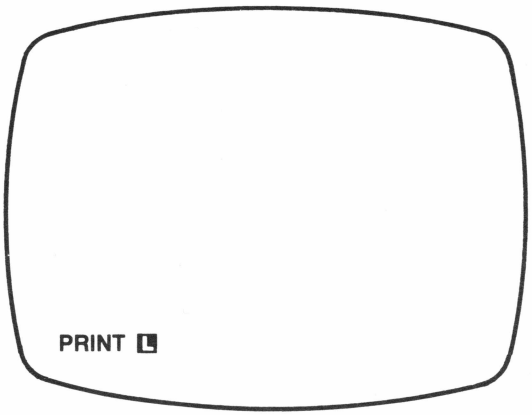
makes the computer compute the sum 2+2 and display (**PRINT**) the result on the screen. An instruction like this, which the computer acts upon immediately, is technically called a BASIC *command*.

(There are other BASIC *statements* that are used in programs — they are not operated on immediately. These will be covered in Chapters 4-7.)

To type in this command,

1. First type **PRINT**. *But*, although as you can see the keyboard has a key for each letter, you do not spell the word out P,R,I,N,T. As soon as you press P,

the whole word will come up on the screen, followed by a space, and the screen will look like this:



The reason is that at the beginning of each command the computer is expecting a *keyword* — a word that specifies what kind of command it is. The keywords are written above the keys, and you will see that "**PRINT**" appears above the P key, so that to get "**PRINT**" you have to press P.

The computer lets you know that it expects a keyword by the K that you had to start off with. There is almost always a white-on-black (*inverse video*) letter, either K or L (or, we shall see later, F or G), called the *cursor*. The K means "Whatever key you press, it will be taken as a keyword." As you saw, after you had pressed P for **PRINT**, the K changed to an L .

This system of pressing just one key to get more than one symbol is used a lot on the T/S 1ØØØ. In the rest of the manual, words with their own keys are printed in **BOLD TYPE**.

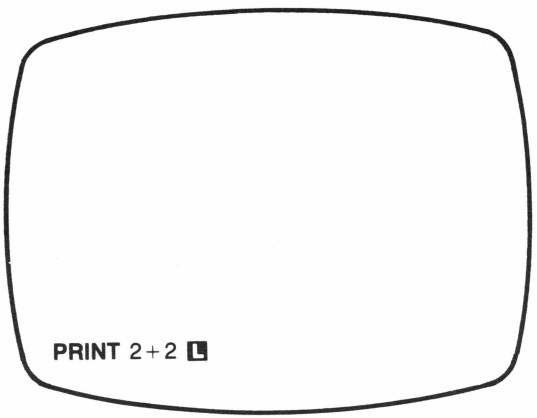
Keywords cannot be typed in letter by letter.

2. Now type 2. This should cause no problem. Again, you should see 2 appear on the screen, and the L move one place to the right.

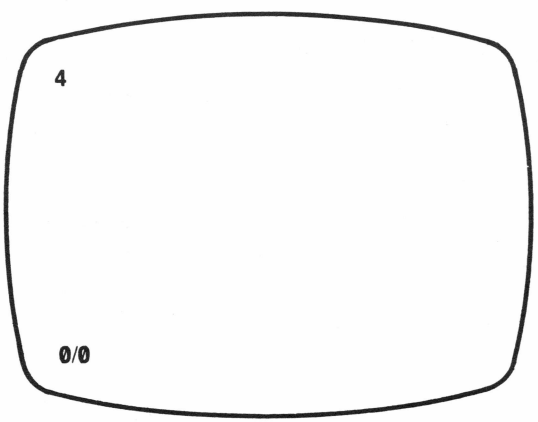
Note also how a space is automatically put between **PRINT** and 2. This is done as much as possible, so that you hardly ever have to type a space. If you do type a space, it will appear on the screen, but it will not affect the command.

3. Now type +. This is a *shifted* character (such characters are marked in red — the color of **SHIFT** itself — in the top right-hand corner of each key), and to get "+" you must hold down the key SHIFT , and, while doing that, press the key K+ (LIST / LEN) .

4. Now type 2 again. The screen will look like this:



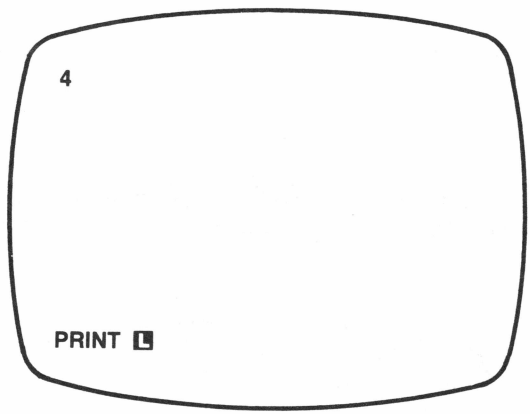
5. Now press **ENTER**, the key FUNCTION ENTER . You press this key when you are finished with a line. The computer will then compute. In our case, the screen changes to



4 is the answer — but of course you don't need a computer to figure that out.

Ø/Ø is the *report* in which the computer tells you how it got where it is. (Note that zero is written with a slash to distinguish it from capital O. This is standard notation in computing.) The first Ø means "OK, no problems." (At the very end of the book, where you can flip to it conveniently, there is a list of other report codes that can arise; for instance, if something goes wrong.) The second Ø means that "the last thing done was line Ø." You will see later — when you start to write programs — that a statement can be given a number and stored away for execution later: it is then a *program line*. Commands not in programs do not actually have numbers, so for the sake of reports the computer treats them as line Ø.

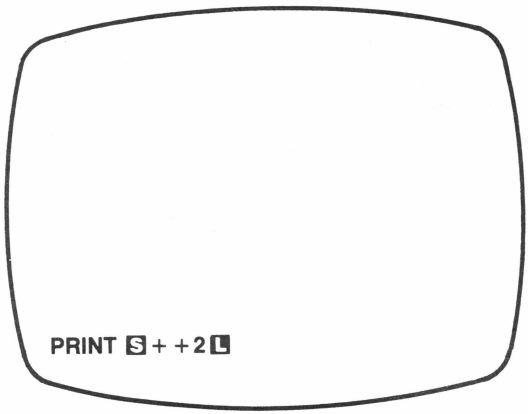
You should imagine a report as hiding a K cursor — if you press P for **PRINT** now, the report will disappear and the screen will change to



The cursor can also be used for correcting mistakes: type + +2, to get

PRINT + +2 L

on the bottom line. When you press **ENTER** you get



The S is the *syntax error marker* (the syntax is the grammar of messages, saying which are allowed and which are not); it shows that the computer got as far as "PRINT," but after that it was not a "legal" message.

What you want to do, of course, is to delete the first + and replace it with — let us say — 3. First you have to move the cursor so that it is just to the right of the first +; there are two keys, ◊ and ◊ (shifted 5 and shifted 8), that move the cursor left and right. Holding **SHIFT** down, press the ◊ key twice. This moves the cursor left two places to give you

PRINT +L+2

Now press the **DELETE** key (shifted Ø), and you will get

PRINT L+2

DELETE removes the character (or keyword) immediately to the left of the cursor.

If you now press 3 a "3" will be inserted, again immediately to the left of the cursor, giving

PRINT 3L+2

and pressing **ENTER** gives the answer (5).

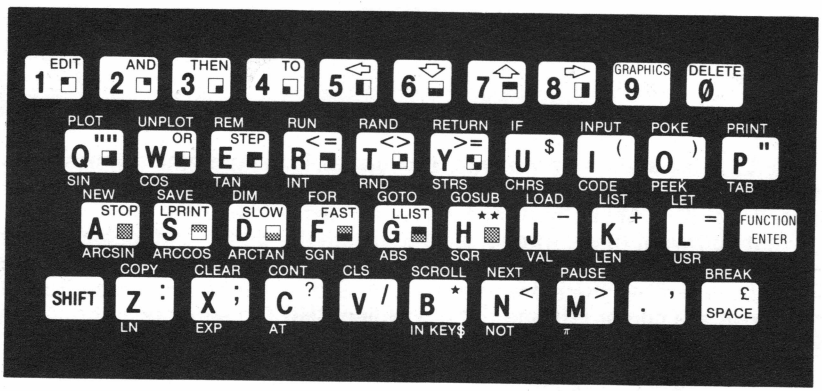
The ◊ key (shifted 8) works just like the ◊ key, except that it moves the cursor right instead of left.

Summary

This chapter has covered how to type in commands for the T/S 1000.
explaining the single keystroke system for words,
the K and L cursors,
reports
the syntax error marker, S
and how to correct mistakes using ◊ , ◊ and **DELETE**.

The Keyboard

Here is a picture of the keyboard.



If you are not familiar with typewriters, you can use this illustration to learn where all the letters are. Remember that to use **SHIFT**, you have to hold it down at the same time that you press another key. Do not confuse the digit Ø with letter O.

If you *are* a typist, you must also remember that you cannot use a lower case L for the numeral 1.

Chapter 4

Writing a Program

We are now going to write our first computer *program*, which is a set of instructions to the computer. In the last chapter, we gave the computer *commands*, which it executed immediately. Now we will begin to construct *statements* which the T/S 1000 will carry out in order. Although we will start small, there are programs of many thousands of lines, which direct computers to carry out lengthy and complicated procedures; the power of computing is in the ability of the machine to receive, store and carry out many different complex programs.

We will program the T/S 1000 in a computer "language" called BASIC (which stands for Beginner's All-purpose Symbolic Instruction Code). Invented at Dartmouth College, BASIC looks more like English than other computer languages and can be used in an *interactive* mode, where the user types in something and the computer responds immediately via a display like your TV screen. When you have learned to program the T/S 1000, you will find that you can use programs in BASIC drawn from other books and magazines. You will almost certainly have to alter them slightly to make them run on the T/S 1000, because every machine has its own slightly different version of BASIC.

Let's get started. First, type **NEW** and then **ENTER**. That erases anything

that might be left over in the computer from Chapter 3 and clears the decks for action.

Then, type

```
10 PRINT "HELLO, JACK"
```

If you are a typist, be sure you use numeral 1, not the lowercase L, in the 10 (for one thing, there is no lower case on the T/S 1000. . .). Also, be sure you use zero (Ø with the slash mark through it) rather than the letter O. The 10 in front of the same command you typed before makes it a program line. Type **ENTER** and see what happens.

Now the cursor is ready again at the bottom of the screen. Type this:

```
20 GOTO 10          ENTER
```

Note that **GOTO** is a keyword (over the G key) and should not be spelled out.

Now you have a program! To execute it, type **RUN** (keyword over R: notice how most of the keywords are over the letters they start with?) and **ENTER**.

How about that? That's a lot of stuff for the little effort you put in. As we said, the computer is not very smart. But it is *fast*, and accurate, and tireless with doing repetitions. One of the most powerful commands in BASIC is the **GOTO**, directing the computer to go back to an earlier line and repeat what it's done before.

You'll notice the report 5/10 at the bottom of the screen. That means it stopped because the screen was filled (5), and last executed line 10. You can clear the screen and allow the computer to continue by typing **CONT**, then **ENTER**.

Here's something else. While the computer is running down the screen saying "Hello, Jack," press the **BREAK** key (which doubles as the **SPACE** key) and notice that the list stops wherever it is when you pressed **BREAK**. The computer checks at the end of every program line to see if anyone has pressed **BREAK**; if so, it stops the program. (You can also use **BREAK** to stop a runaway program, an "endless loop," or a misfired **LOAD**ing from a tape cassette. If **BREAK** doesn't work, you wind up having to pull the plug and you lose whatever's in the machine.)

You can restart the program after the **BREAK**, also by keying **CONT**, then **ENTER**. **CONT**inue lets you continue when the screen is full, when you interrupt the program with **BREAK** or **STOP**, or when a program interrupts with the **STOP** command.

Okay. After you've had your fill of fooling around with **BREAK** and **CONT**, let the program stop with a full screen, and touch **ENTER** again. Your program is back on the screen. You can **RUN** it again (go ahead: **RUN**, **ENTER**) and then display it again (**ENTER**). You can go on to something else and keep it in the computer until you erase it, the computer's memory gets full, or you unplug the machine.

Chapter 5

Punctuation and Arranging Output on the Screen

You can do wonders with punctuation marks in a T/S 1ØØØ Basic program. For instance, let's clean up with **NEW**, **ENTER**, then type

```
1Ø PRINT "JACK",
2Ø GOTO 1Ø
```

Notice that we've decided to dispense with the "HELLO". And, instead of JACK, why don't you go ahead and use your own name? In a program, the computer will treat everything in quotes as a *string* (a string of characters), and the string can be of any length — and it doesn't have to make sense. The T/S 1ØØØ will print JACK, but it will just as faithfully print HERMAN, JACQUELINE, or KLJY66??F.

Most importantly, notice the comma we've added — and to the dismay of your sixth-grade English teacher, it is *after* and *outside* the quotation marks. Type **RUN**, **ENTER**.

The screen is 32 characters wide in T/S 1ØØØ operations, numbered Ø to 31, and the comma moves the beginning of the next entry to position 16 or position Ø (on the next line), depending on where the last entry finished. So you can make two columns, as long as your name is not more than 16 letters long.

Let's see what a semicolon does. But don't wipe out the program with **NEW**. Just press **ENTER** to bring it back.

There it is, at the top of the screen. There is a different kind of cursor up there, ▶ ,showing that line 2Ø was the last one you entered. Now hold **SHIFT** down and press the *up* arrow, ⇦ , (shifted 7), moving the cursor up to line 1Ø. Now hold **SHIFT** and press **EDIT** (shifted 1), and line 1Ø (the one with the cursor) comes down to the bottom of the screen where you can work on it. Now use the *right* arrow, ⇨ , (shifted 8), to move the cursor all the way to the end of the line, then **DELETE** (shifted Ø) to erase the comma. Type in the semicolon instead (shifted X) and **ENTER**. The edited line replaces the old one in the program. **RUN** the program.

The semicolon starts the second **PRINT** entry right after the first one. But that's pretty crowded. We need to add a space. Press **ENTER** to get the program listing back. Use the **EDIT** command and the cursor arrows to bring line 1Ø down and add a space between the K in JACK (or the last letter in your name) and the quotation mark. Just position the cursor between K and " and press **SPACE**. Then **ENTER**. And **RUN**. And **ENTER**. Behold.

There's a trick you can use with the **EDIT** function, by the way. If you're working on a long program line and have it all messed up, so that you wish it would go away so you can start over, you can press **DELETE** as many times as it takes to get back to the left margin. Or you can just press **EDIT**, bringing down your last successful program line and wiping out the one you were working on. Then hit **ENTER** and return the un**EDIT**ed line to the program, giving you a clean slate to work with on the bottom!

We still have one problem with JACK, and most likely with your name, too — the name is divided in different ways at the ends of the lines. Suppose we want to line up the names in neat columns. With a comma we can get two columns; how can we get more?

With the **TAB** command, which is actually a *function*. (Well, actually it's not a function, but it's treated like one because of where it is located on the keyboard.)

Clear the program (**NEW**, **ENTER**) and type

```
1Ø PRINT
```

You'll then find **TAB** written under the P key. To get it (and to get all other functions written under keys), hold down **SHIFT** and press **FUNCTION** (shifted **ENTER**). The cursor changes to ▛ , meaning that the T/S 1ØØØ will interpret the next key as a function. (You can let go of the **SHIFT** now, and the cursor is still ▛ .) Press **TAB**. It appears on the program line and the cursor changes back to ▌ . (You can't have two functions in a row without something for them to operate on, so the computer automatically switches out of the function mode.) Type in a 1 and a semicolon, the "JACK" or your name. So far the line looks like this (but don't **ENTER** it; we're not done yet):

```
10 PRINT TAB 1;"JACK"
```

Now we want to separate the columns of names by a space. Since JACK has four letters, we add a space by making the next entry at **TAB** 6, using semicolons between each entry:

```
10 PRINT TAB 1;"JACK";TAB 6;"JACK"
```

If your name has a different number of letters from JACK, just add the number of letters, plus one for the space, to the last **TAB** position in order to get the next **TAB** position. Keep doing this, but be sure you don't have more than 31 letters or spaces in all, however many times you put your name in (if you do, just eliminate one entry, using **DELETE**, the cursor arrows and **EDIT** if necessary):

```
10 PRINT TAB 1;"JACK";TAB 6;"JACK";TAB 11;
"JACK";TAB 16;"JACK";TAB 21;"JACK";TAB 26;"JACK"
```

Then **ENTER**. From now on, we'll stop reminding you about **ENTER**. **ENTER** every line when you're done with it. Type

```
20 GOTO 10
```

RUN the program. Isn't that tidy?

Chapter 6

Loops and Ifs

We mentioned earlier that **GOTO** could be a very powerful command, because it directs the computer to repeat an action over and over. This is called *looping*, and the portion of the program that is repeated is called a *loop*. Generally, if you don't have a way of ending repetitions (other than **BREAK** or pulling the plug), you have what is called an "endless loop," which is messy programming practice. Normally you want the action repeated a specific number of times (maybe twice, maybe ten thousand times — it's all the same to a computer). You can use **GOTO** and use a *counter* or *control variable*. Type this:

```
10 LET I=1
```

The **LET** statement *assigns* a value to a *variable*. In this case, I is the *name* of the variable, and it is being assigned the *value* of 1. (You can use any number of letters or digits in a variable name in T/S 1000 BASIC as long as the first one is a letter.) Sometimes you spell out a word like RENTPAYMENT for clarity; often you use just one letter for speed. You can only assign numerical values to regular variables, but you can assign anything in quotes to

string variables, which are always named by a single letter followed by a dollar sign (shifted U). In the previous program, if your name was ABERCROMBIE, instead of spelling it out each time, you might have done this:

```
10 LET A$="ABERCROMBIE"
20 PRINT TAB 1;A$;TAB 13;A$;...and so on)
```

Let's go on with our program for this chapter. Now type:

```
20 PRINT "HELLO, JACK"
30 LET I=I+1
```

Hold on right there, partner! You expect me to believe that I=I+1? What kind of math is that?

Well, it isn't math, it's programming. And in a **LET** statement in BASIC, it's okay. It means "set the new value of I equal to the old value plus 1"; in other words, increase the value of I by 1. So each time we go through the program, in a repeating loop, the counter will be raised by 1. In a moment we'll see why. Type on:

```
40 IF I=6 THEN GOTO 60
```

Aha! So, each time we get to line 40, the T/S 1000 will check the value of I. When it reaches 6 — the sixth time through the program — it will **GOTO** line 60. Until I equals 6, the program will just go on to the next line after 40 instead.

```
50 GOTO 20
60 STOP
```

If I=6, we **GOTO** 60 and **STOP**. If I is not equal to 6, we go past 40 to 50, which in turn tells us to **GOTO** 20 and start over from there. Let's **RUN** it.

We've printed "HELLO JACK" five times (the sixth time we went to 60 instead). Press **ENTER** to get back the listing. The **IF** statement, with its decision-making ability, is another of BASIC's important features.

But that was a fairly clumsy loop. There is a cleaner approach, called the **FOR/NEXT** loop. Type this in *without* first erasing the program above:

```
100 FOR I=1 TO 5
```

(Use shifted 4 for **TO**; don't spell it out)

```
110 PRINT "HELLO, JACK"
120 NEXT I
```

(I is commonly used as the counting or control variable, incidentally; control variables of **FOR/NEXT** loops must have names a single letter long.)

To **RUN** that part of the program, type in **RUN** 100 instead of just **RUN** (the computer is saving all the lines and will start running at any line you choose). After you've seen how that three-line program does just what the previous six-liner does, look at the listing again by typing **LIST** 100. If you want to see both listings (or, more properly, all of what is really a single listing), just hit **ENTER** again — or **LIST** with no line number, and then

ENTER. (This is the way most BASICs handle the listing function; the T/S 1000's recalling of a listing with just **ENTER** is unusual.)

SPECIAL NOTE: When you use "**RUN** 100," you clear all variables before beginning the run. If for some reason you want to save variables that you have assigned values to, use "**GOTO** 100" instead. The command **RUN** means, to the T/S 1000, **CLEAR** (which you can also enter from the keyboard if you want to clear all the variables — for instance, if you need the space — but leave the program and the screen intact), then **GOTO**.

An extra subtlety is that the control variable does not have to go up by 1 each time: you can change this 1 to anything else you like by using a **STEP** part in the **FOR** statement. If line 100 read

```
100 FOR I=1 TO 5 STEP 2
```

then you'd get 3 "HELLO, JACK"s before the loop reached its limit and stopped. The **STEP** need not be in whole numbers, and the control value need not hit the limit evenly — it goes on looping as long as it is less than or equal to the limit.

You must be careful if you are running two **FOR/NEXT** loops together, one inside the other. Try this program, which prints out a complete set of 6-spot dominoes:

```
10 FOR M=0 TO 6
20 FOR N=0 TO M
30 PRINT M;":";N;"   ";
40 NEXT N
50 PRINT
60 NEXT M
```

You can see that the N-loop is entirely inside the M-loop — in other words, they are correctly *nested*. What must be avoided is two **FOR/NEXT** loops that overlap without either being entirely inside the other, like this:

```
10 FOR M=0 TO 6
20 FOR N=0 TO M
30 PRINT M;":";N;"   ";          WRONG!
40 NEXT M
50 PRINT
60 NEXT N
```

Two **FOR/NEXT** loops must be either one inside the other, or completely separate. Another thing to avoid is jumping into the middle of a **FOR/NEXT** loop from outside. (You might do this with a **GOTO** directing the computer to a line after a **FOR** statement, and when you get to the **NEXT** statement, you will probably get an error report 1 or 2.)

Now let's take a look at a program that might be of some use, and some statements we haven't seen before. First type it in, and then we'll analyze it:

```
10 REM PROGRAM TO MULTIPLY BY PI
(Spell out PI in lines 10 and 20)
20 PRINT "THIS PROGRAM MULTIPLIES ANY NUMBER BY PI"
30 PRINT
40 PRINT "ENTER A NUMBER"
50 INPUT A
60 PRINT A;" TIMES PI= ";A*PI
(Use the function PI — the π under the M key — although the computer will
print it to look just like the spelled-out version in lines 10 and 20)
70 PRINT
80 GOTO 30
```

Now, let's talk a bit about this program before we run it.

1. Notice the line numbers we have been using are in multiples of ten. Always do this in a "first draft." Because the computer automatically puts the program lines in numerical order, you can insert a line 15 if you want or need to later — and after that, a 12 and a 13.
2. The keyword **REM**, for "REMARK" or "REMINDER," identifies a line that is printed out in the program listing to help a later user understand the program, but ignored by the computer during a **RUN**.
3. The keyword command **INPUT** in line 50 causes the computer to stop and wait for the user to enter a number. The L cursor at the bottom of the screen when you run the program indicates that the T/S 1000 is waiting for input, but it is best to indicate more clearly to the user that he has to do this. That's the function of line 40.

SPECIAL NOTE: When you wish to **INPUT** a string (as we will later on), you must indicate a string variable in the program — A$ instead of A — and the cursor prompt will appear at the bottom of the screen enclosed in quotation marks. If, at that point, you want to **STOP** the program, you can't just input the word **STOP** — the computer will treat it as a string! You must use the left arrow key to move the cursor to a point outside and before the quotation marks and then type **STOP**.

4. Lines 30 and 70, **PRINT** without anything to be printed, essentially tell the computer to print a blank line, or skip a line. This simply makes the output easier to read on the screen because of the extra space. After you've run the program, you might want to change the listing by typing 30 **ENTER** and 70 **ENTER** (eliminating both lines) and see how much more crowded the screen looks.
5. Another piece of spacing for legibility is found in line 60. Notice that there is a space typed in after the first quotation mark and before the second one. Don't leave them out.

Now **RUN** the program.

You can do the same thing on a pocket calculator with memory (store an accurate value for PI in memory, enter a new number, hit the multiply key and the "memory recall" key for each answer). But even for this trivial piece of math, the computer is easier and handier, and tirelessly guides you through the exercise, reminding you along the way what you are doing. Imagine the value of having the computer do this for you on a more complex task!

Chapter 7

More About IF

All the programs we've seen so far have been fairly predictable — they followed all the instructions from beginning to end, and then maybe went back to the beginning again. This is not all that useful. In practice the computer would be expected to distinguish between different cases and act accordingly; it does this by using the **IF** statement.

Clear the computer (using **NEW**), and type in and run this program:

```
10 PRINT "I AM THINKING OF A"
20 PRINT "NUMBER FROM 1 TO 5"
30 PRINT "CAN YOU GUESS IT?"
40 LET A=INT(RND*5)+1
50 INPUT B
60 IF A=B THEN GOTO 90
70 PRINT "GUESS AGAIN"
80 GOTO 50
90 PRINT "CORRECT"
```

For a discussion of **RND**, see Chapter 15, "Functions", especially exercise 4.

As you can see, an **IF** statement takes the form

IF condition **THEN** statement

The statements here are **GOTO** statements, but they could be anything at all, even more **IF** statements. The condition is something that is going to be proved either true or false. If it comes out as true, the statement after **THEN** is executed; otherwise, it is skipped over.

The most useful conditions compare two numbers or two strings: they can test whether two numbers are equal, or whether one is bigger than the other; and they can test whether two strings are equal, or whether one comes before the other in alphabetical order. They use the relations =, <, >, <=, >= and <>.

=, which we have used twice in the program (once for numbers and once for strings) means "equals". It is not the same as = in a **LET** statement.

< means "is less than," so that

```
        1<2
        -2<-1
and     -3<1
```

all hold (they have the value *true*), but

```
        1<Ø
and     Ø<-2
```

do not (they have the value *false*).

To see how this works, let's write a program to input numbers and display the biggest so far.

```
1Ø PRINT "NUMBER","BIGGEST SO FAR"
2Ø INPUT A
3Ø LET BIGGEST=A
4Ø PRINT A,BIGGEST
5Ø INPUT A
6Ø IF BIGGEST<A THEN LET BIGGEST=A
7Ø GOTO 4Ø
```

The crucial part is line 6Ø, which updates BIGGEST if its old value was smaller than the new input number A.

> (shifted M) means "is greater than" and is just like < except in reverse. You can distinguish between them by remembering that the thin end points to the number that is supposed to be smaller.

<= (shifted R — do not type it as < followed by =) means "is less than or equal to," so that it is like < except that it holds even if the two numbers are equal: thus 2<=2 holds, but 2<2 does not.

>= (shifted Y) means "is greater than or equal to" and is similarly like >.

<> (shifted T) means "is not equal to," the opposite in meaning from =.

Mathematicians usually write <=, >= and <> as $\leq$, $\geq$ and $\neq$. They also write such sequences as "2<3<4" to mean "2<3 and 3<4," but this is

not possible in BASIC.

These relations can be combined by using the *logical operations* **AND**, **OR** and **NOT**.

one relation **AND** another relation

holds whenever both relations hold.

one relation **OR** another

hold whenever one of the two relations does (or both do).

NOT relation

holds whenever the relation does not.

Logical expressions can be made with relations and **AND**, **OR** and **NOT**, just as numerical expressions can be made with numbers and +, − and so on; you can even put in parentheses if necessary. **NOT** has priority 4, **AND** 3 and **OR** 2.

To illustrate, clear the computer and try this program, expanded from the one at the beginning of the chapter.

```
10 PRINT "I AM THINKING OF A"
20 PRINT "NUMBER FROM 1 TO 5"
25 PRINT "CAN YOU GUESS IT?"
30 PRINT
35 LET T=0
40 LET A=INT(RND*5)+1
50 INPUT B
55 LET T=T+1
60 IF A<>B AND T>=3 THEN GOTO 100
65 IF A=B THEN GOTO 90
70 PRINT "GUESS AGAIN"
80 GOTO 50
90 PRINT "CORRECT"
95 STOP
100 PRINT "SORRY, GAME OVER"
110 PRINT "YOU DID NOT GET IT IN"
120 PRINT "THREE GUESSES."
130 PRINT "THE ANSWER WAS  ";A
```

Notice the **STOP** statement in line 95. Otherwise, when the program reached line 90 (because the answer was correct), it would then go on to 100 and say the right answer had *not* been given. This would be confusing, to say the least.

Lastly, we can compare not only numbers but also strings. We have seen how =, <, >, <>, <=, and >= work when comparing numbers.

What does "less than" mean for strings? One thing it does not mean is "shorter than" so don't make that mistake. We make the distinction that one string is less than another if it comes first in alphabetical order: thus

"SMITH"	<	"SMYTHE"
"SMYTHE"	>	"SMITH"
"BILLION"	<	"MILLION"
"DOLLAR"	<	"POUND"

all hold. <= means "is less than or equal to," and so on, just as for numbers.

Note: In some versions of BASIC — but not on the T/S 1000 — the **IF** statement can have the form

IF condition **THEN** line number

This means the same as

IF condition **THEN GOTO** line number

You must use both **THEN** and **GOTO** in Sinclair BASIC.

Summary

Operations: =, <, >, <=, >=, <>, **AND**, **OR**
Functions: **NOT**

Exercises

1. <> and = are *opposites* in the sense that **NOT** A=B is the same as A<>B and

NOT A<>B is the same as A=B

Prove to yourself that < and >=, and > and <= are opposites in the same way, so that you can always remove **NOT** from in front of a relation by changing the relation.

Also,

NOT (a first logical expression **AND** a second)

is the same as

NOT (the first) **OR NOT** (the second)

and **NOT** (a first logical expression **OR** a second)

is the same as

NOT (the first) **AND NOT** (the second).

Using this, you can work **NOT**s through parentheses until eventually they are all applied to relations, and then you can dispose of them. Thus, logically

speaking, **NOT** is unnecessary, but you may still find that using it makes a program clearer.

2. BASIC can sometimes work along different lines from English. Consider, for instance, the English clause "if A doesn't equal B or C". How would you write this in BASIC? The answer is *not*

"**IF** A <> B **OR** C" nor "**IF** A <> B **OR** A <> C"

Don't worry if you don't understand exercise 3, 4 and 5, the points covered in them are fairly advanced.

3. (Skip this unless you already know BASIC thoroughly.)
Try

PRINT 1=2,1<>2

which you might expect to give a syntax error. In fact, as far as the computer is concerned, there is no such thing as a logical value.

(i) =, <, >, <=, >= and <> are all number-valued binary operations, with priority 5. The result is 1 (for true) if the relation holds, and Ø (for false) if it does not.

(ii) in

IF condition **THEN** statement

the condition can actually be any numerical expression. If its value is Ø, then it counts as false and any other value counts as true. Thus the **IF** statement means exactly the same as

IF condition <> Ø **THEN** statement

(iii) **AND**, **OR** and **NOT** are also number-valued operations.

X **AND** Y has the value	X if Y is non-zero (counting as true) Ø if Y is zero (counting as false)
X **OR** Y has the value	1 if Y is non-zero X if Y is zero

and

NOT X has the value	Ø if X is non-zero 1 if X is zero

With this in mind, read through the chapter again, making sure it all works.

In the expression X **AND** Y, X **OR** Y and **NOT** X, X and Y will each usually take the value Ø or 1, for false or true. Work out the ten different combinations and see if they do what you expect **AND**, **OR** and **NOT** to do.

4. Try this program:

```
10 INPUT A
20 INPUT B
30 PRINT (A AND A>=B)+(B AND A<B)
40 GOTO 10
```

Each time, it prints the larger of the two numbers A and B. Why? Convince yourself that you can think of

X **AND** Y

as meaning

'X if Y (else the result is Ø)'

and of

X **OR** Y

as meaning

'X unless Y (in which case the result is 1)'

An expression using **AND** and **OR** in this way is called a *conditional* expression. An example using **OR** could be

```
LET RETAIL PRICE=PRICE LESS TAX*(1.05 OR V$="NON-
TAXABLE")
```

Notice how **AND** tends to go with addition (because its default value is Ø), and **OR** tends to go with multiplication (because its default value is 1).

5. You can also make string-valued conditional expressions, but only using **AND**.

X$ **AND** Y has the value X$ if Y is non-zero
"" if Y is zero

so it means "X$ if Y (else the empty string)."

Try this program, which inputs two strings and arranges them in alphabetical order.

```
10 INPUT A$
20 INPUT B$
30 IF A$<=B$ THEN GOTO 70
40 LET C$=A$
50 LET A$=B$
60 LET B$=C$
70 PRINT A$;" ";("<" AND A$<B$)+
("=" AND A$=B$);" ";B$
80 GOTO 10
```

6.Try this program:

```
10 PRINT "X"
20 STOP
30 PRINT "Y"
```

When you run it, it will display "X" and stop with report 9/20. Now type

CONT

You might expect this to behave like "**GOTO** 20," so that the computer would just stop again without displaying "Y"; but this would not be very useful, so the program is arranged so that for reports with code 9 (**STOP** statement executed), the line number is increased by 1 for a **CONT** statement. Thus, in our example, "**CONT**" behaves like "**GOTO** 21" (which, since there are no lines between 20 and 30, behaves like "**GOTO** 30").

7. Many versions of BASIC (but not the Sinclair BASIC) have an ON statement, which takes the form

ON numerical expression GOTO line number, line number, . . ., line number.

In this the numerical expression is evaluated; suppose its value is n then the effect is that of

GOTO the nth line number

For instance,

ON A GOTO 100, 200, 300, 400, 500

Here, if A has the value 2, then 'GOTO 200' is executed. In Sinclair BASIC this can be replaced by

GOTO 100*A

In case the line numbers don't go up neatly by hundreds like this, work out how you could use

GOTO a conditional expression

instead.

Chapter 8

SLOW and FAST

The T/S 1ØØØ can run at two speeds — SLOW and FAST. When it is first plugged in, it runs in the SLOW mode and can compute and display information on the screen simultaneously. This mode is ideal for "moving graphics" because it gives priority to maintaining the display and does its computing during the periods when the television is making the blank parts of the picture at the top and bottom of the screen.

However, it can go roughly four times as fast, which it does by giving priority to the computation and only maintaining the picture when it has nothing else to do. To see this working, type

FAST

Now whenever you press a key, the screen will blink — this is because the computer has stopped displaying a picture while it determined what key you pressed.

Type in a program, say,

```
10 FOR N=0 TO 255
20 PRINT CHR$ N;
30 NEXT N
```

Note that **CHR$** is a *function*, as we mentioned briefly in Chapter 4. Press **SHIFT** and **ENTER** to get the F cursor, then type U to get **CHR$**, which is under the U.

When you run this, the whole screen will become an indeterminate gray until the end of the program, when the output from the **PRINT** statement will come up on the screen.

The picture is also displayed during an **INPUT** statement, while the computer is waiting for you to type the **INPUT** data. Try this program:

```
10 INPUT A
20 PRINT A
30 GOTO 10
```

To get back into normal (compute and display) mode, type

```
SLOW
```

It will often be just a matter of taste whether you want compute and display mode for continuity on the screen, or fast mode for speed; but in general you will use the fast mode

(i) when your program contains a lot of numerical calculation, especially if it doesn't print much — time doesn't seem to drag quite so much if you can see output coming up on the screen fairly frequently.

(ii) when you are typing in a long program. You will already have noticed that the listing gets redone every time you enter a new program line, and this can be annoying.

You can use **SLOW** and **FAST** statements within programs.

For example,

```
10 SLOW
20 FOR N=1 TO 64
30 PRINT "A";
40 IF N=32 THEN FAST
50 NEXT N
60 GOTO 10
```

Summary

Statements: **FAST, SLOW**

Chapter 9

Subroutines

Sometimes different parts of your program will have similar jobs to do, and you will find yourself typing in the same lines two or more times. This is not necessary. You can type the lines in once, in the form known as a *subroutine*, and then use, or *call*, them anywhere else in the program without having to type them in again.

To do this, use the statements **GOSUB** (Go to SUBroutine) and **RETURN**.

GOSUB n

where n is the line number of the first line in the subroutine, is just like **GOTO** n except that the computer stores the line number of the **GOSUB** statement so that it can come back again after doing the line. It does this by putting the line number (the *return address*) on top of a pile of statements (the *GOSUB stack*).

RETURN

takes the top line number off the **GOSUB** stack and goes on to the line after it.

As an example,

```
10 PRINT "THIS IS THE MAIN PROGRAM"
20 GOSUB 1000
30 PRINT "AND AGAIN";
40 GOSUB 1000
50 PRINT "AND THAT IS ALL"
60 STOP
1000 REM SUBROUTINE STARTS HERE
1010 PRINT "THIS IS THE SUBROUTINE"
1020 RETURN
```

Without the **STOP** statement in line 60 the program would run on into the subroutine and cause error 7 when the **RETURN** statement was reached.

As another example, suppose you want to write a computer program to handle yards, feet and inches. You will have three variables Y, F and I (and maybe others — Y1, F1, I1, and so on). The arithmetic is easy. First you do it separately on the yards, feet and inches — for instance, to add two quantities of distance, you add the inches, add the feet, and add the yards; to double the distance, you double the inches, double the feet, and double the yards. Then adjust the quantities to the correct form so that the inches are between 0 and 12 and the feet are between 0 and 3. This last stage is common to all operations, so we can make it into a subroutine.

Putting aside the notion of subroutines for a moment, it is worth your while to try to write the program yourself. Given the arbitrary numbers Y, F, and I, how do you convert them into the correct number of yards, feet, and inches?

What first comes to mind will probably be something like 1Y..4F..14I which you want to convert to 2Y..2F..2I. This is not so difficult. But suppose you have negative numbers. Let's go back to our initial values 1Y..4F..14I; negatively they become −1Y..−4F..−14I, which would then turn out to be −2Y..−2F..−2I. And what about fractions? If you divide 3Y..1F..7I by two, you get 1.5Y..0.5F..3.5I; and although this has the feet between 0 and 12, and the yard as 1.5, it is certainly not as good as 1Y..2F..3.5I. Try to work out your own answers to all this and use them in a computer program — before you read any further.

Here is one solution, using functions which will be described more thoroughly in Chapter 15. Chapter 5 tells you how to program them.

```
1000 REM SUBROUTINE TO ADJUST Y, F, I TO THE NORMAL
FORM FOR YARDS, FEET, AND INCHES
1010 LET I=36*Y+12*F+I
1020 REM NOW EVERYTHING IS IN INCHES
1030 LET E=SGN I
1040 LET I=ABS I
1050 REM WE WORK WITH I POSITIVE, HOLDING
ITS SIGN IN E
1060 LET F=INT (I/12)
1070 LET I=(I-12*F)*E
1080 LET Y=INT (F/3)*E
1090 LET F=F*E-3*Y
1100 RETURN
```

On its own this is not much use, because there is no program to set up Y,F and I beforehand, or to do anything with them afterwards. Type in this main program, and also another subroutine, to print out Y, F, and I.

```
10 INPUT Y
20 INPUT F
30 INPUT I
40 GOSUB 2000
45 REM PRINT THE VALUES
50 PRINT TAB 12;"=";
60 GOSUB 1000
65 REM THE ADJUSTMENT
70 GOSUB 2000
75 REM PRINT THE VALUES
80 PRINT
90 GOTO 10

2000 REM SUBROUTINE TO PRINT Y, F, AND I
2010 PRINT " ";Y;"Y..";F;"F..";I;"I";
2020 RETURN
```

Clearly we have saved on program length by using the printing subroutine at 2000, but the adjustment subroutine in fact makes the program longer — by a **GOSUB** and a **RETURN**. Still, program length is not the only consideration. Used with skill, subroutines can make programs easier to understand.

The main program is made simpler by the fact that it uses more powerful statements: each **GOSUB** represents some complicated BASIC. But you can forget that; only the net result matters. Because of this, it is much easier to grasp the main structure of the program.

The subroutines, on the other hand, are simplified for a very different reason, namely that they are shorter. They still use the same old plodding **LET** and **PRINT** statements, but they have to do only a part of the whole job and so are easier to write.

The skill lies in choosing the level — or levels — at which to write the

subroutines. They must be big enough to have a significant impact on the main program, yet small enough to be significantly easier to write than a complete program without subroutines. These examples (*not* recommended) illustrate.

First,

```
  10 GOSUB 1000
  20 GOTO 10

1000 INPUT Y
1010 INPUT F
1020 INPUT I
1030 PRINT " ";Y;"Y..";F;"F..";I;"I";TAB 12;"= ";
1040 LET I=36*Y+12*F+I
  :    :
  :    :
2000 RETURN
```

and second,

```
  10 GOSUB 1010
  20 GOSUB 1020
  30 GOSUB 1030
  40 GOSUB 1040
  50 GOSUB 1050
  :    :
  :    :
 300 GOTO 10

1010 INPUT Y
1015 RETURN
1020 INPUT F
1025 RETURN
1030 INPUT I
1035 RETURN
1040 PRINT " ";Y;"Y..";F;"F..";I;"I";TAB 12;"= ";
1045 RETURN
1050 LET I=36*Y+12*F+I
1055 RETURN
  :    :
  :    :
```

The first, with its single powerful subroutine; and the second, with its many trivial ones, demonstrate quite opposite extremes, but with equal futility.

A subroutine can call another, or even itself (a subroutine that calls itself is *recursive*), so don't be afraid of having several layers.

Summary

Statements: **GOSUB, RETURN**

Exercises

1. The example program is virtually a universal distance calculator. How would you use it
   (i) to convert yards and inches into yards, feet and inches?
   (ii) to convert meters into yards and feet?
   (iii) to find fractions of a yard? (e.g., a third of a yard, or a foot.)

   Put in a line to round inches off to the nearest inch.

2. Add two statements to the program:

```
4 LET ADJUST=1000
7 LET YFIPRINT=2000
```

   and change

```
GOSUB 1000    to    GOSUB ADJUST
GOSUB 2000    to    GOSUB YFIPRINT
```

   This works exactly as you'd hope; in fact, the line number in a **GOSUB** (or **GOTO** or **RUN**) statement can be any numerical expression. (This may not work on computers other than the T/S 1000, because it is not standard BASIC.)
   This kind of thing can work wonders for the clarity of your programs.

3. Rewrite the main program in the example to do something quite different, but still using the same subroutines.

4. ..**GOSUB** n
   ..**RETURN**

in consecutive lines can be replaced by

..**GOTO** n

Why?

5. A subroutine can have several *entry points*. For instance, because of the way our main program uses them, with **GOSUB** 1000 followed immediately by **GOSUB** 2000, we can replace our two subroutines by one big one that adjusts Y, F and I and then prints them. It would have two entry points: one at the beginning for the whole subroutine, and another further on for the printing part only.
   Make the necessary rearrangements.

6. Run this program:

```
10 GOSUB 20
20 GOSUB 10
```

The return addresses are pushed on to the **GOSUB** stack in droves, but they never get taken off again; and eventually there is no room for any more in the computer. The program then stops with error 4 (see Report Codes).

You might have difficulty in clearing them out again without losing everything, but this will work.

(i) Delete the two **GOSUB** statements.

(ii) Insert two new lines

```
11 RETURN
21 RETURN
```

(iii) Press

RETURN

The return addresses will be stripped off until you get error 7.

(iv) Change your program so the same thing doesn't happen again.

How does this work?

Chapter 10

When the Computer Gets Full

The T/S 1ØØØ has only limited internal storage, and it is not hard to fill it. The best indication that this has happened is usually an error report 4, but other things can happen, and some of them are rather strange. If you have a RAM pack, detach it for a moment.

The *display file* — the area inside the computer where it stores the television picture — is designed so that it only takes up space for what has been printed so far: a line in the display consists of up to 32 characters and then an **ENTER** character. This means that you can run out of room by printing something, and the most obvious time is while making a listing. Type

```
NEW
DIM A(355)
1Ø FOR I=1 TO 15
2Ø PRINT I
```

(The **DIM**ension statement sets aside space in the computer's memory, as we will see in Chapter 17. For now, we are using it as a quick way to "use

up'' the memory for this chapter's demonstration. Type it in and don't worry about it.)

Here comes the first surprise: line 1Ø disappears from the listing. The listing must include the current line, 2Ø, and there is no room for both lines. Now type

```
3Ø NEXT I
```

Again, there is only room for line 3Ø in the listing. Now type

```
4Ø REM X          (without ENTER)
```

and you will see line 3Ø disappear and line 4Ø jump to the top of the screen. It has not been entered in the program — you still have the L cursor and can move it about. All you have seen is some obscure mechanism that gives the bottom half of the screen 24 lines to give it priority over the top half. Now type

```
XXXXXX            (still without ENTER)
```

and the cursor will disappear — there is no room to display it. Type another X, without **ENTER**, and one of the Xs will disappear. Now type **ENTER**. Everything will disappear, but the program is still in the computer, as you can prove by deleting line 1Ø and using ◇ and ◇ Now type

```
1Ø FOR I=1 TO 15
```

again — it will move up to the top of the screen as line 4Ø did. But when you press **ENTER**, it will not be entered, although there is no error message or S marker to say that anything is wrong. This is the result of there being no room to check the syntax of a line, and it usually happens only for lines that contain numbers (other than the line number at the beginning).

The solution is to make space somehow, but first delete the line 1Ø that won't go in. Press **EDIT**: the screen will go blank, because there is no room to bring the line down.

Press **ENTER** and you will get part of the listing back. Now delete the line 4Ø (which you really didn't want anyway) by typing

```
4Ø                          (and ENTER)
```

Now try typing in line 1Ø again — it still won't go. Delete it again. You must still find extra space somewhere. Bear in mind that the reason line 1Ø was rejected was probably that there was no room to check the syntax of the two numbers, 1 and 15: so if you delete line 2Ø in the program, you may have room to enter line 1Ø and still have room to reenter line 2Ø (which contains no number) afterwards. Try this. Type

```
2Ø
1Ø FOR I=1 TO 15
2Ø PRINT I
```

and the program is entered properly.

Type

GOTO 1Ø

and again you will find that this line is rejected because its syntax cannot be checked; however, if you delete and type

RUN

it will work. (**RUN** clears out the array, making plenty of space.)

Now type in the same as before from **NEW** up to line 3Ø, and then

4Ø **REM** XXXXXXXXXXX

(11Xs), which will end up looking like 4Ø RE. When you press **ENTER**, the listing will consist only of line 3Ø, and in fact line 4Ø will have been completely lost. This is because it was simply too long to fit in the program. The effect is even worse when the line is a lengthened version of a line that is already in the program, for you will lose both the old line from the program and the new line that was to replace it.

The solution for this is to buy a RAM pack, which fits on the back of the computer.

The T/S 1Ø16 16K RAM pack gives the computer eight times as much *memory* (computer jargon for the storage) as it has in its unexpanded form.

The behavior with the 16K RAM pack is different, because the display file is filled with spaces to make each line 32 characters long (note that **SCROLL** upsets this — see the chapter on Organization of Storage). Now printing and listing will not make the computer run out of memory, and you will not see all the shortened listings and jumping around; but you will still see the lines sticking or getting lost, and again the solution is to find spare space.

If you have a memory expansion board (16K RAM Pack), put it on and go through the typing in this chapter, using

DIM A(3Ø69)

to replace **DIM** A(355).

To summarize

1. If the listing is only partially shown or things start jumping around, the space is getting tight.
2. If **ENTER** seems to have no effect at the end of a line, there is probably no room to deal with a number. Delete the line, using **EDIT-ENTER** or **DELETE**.
3. **ENTER** might lose a line altogether.

For all these oddities, the solution is the same: Don't panic, and look for some spare space.

The first thing to consider is **CLEAR**. If you have variables and you do not mind losing any of them, this is the thing to do.

Failing this, look for unnecessary statements in the program, such as **REM** statements, and delete some of them.

Summary

When the memory fills up odd things can happen; but they are not usually fatal.

Chapter 11

Mathematics with the T/S 1ØØØ

Turn on the computer. You can now use it as a mathematical calculator, along the lines described in Chapter 3: type **PRINT**, then whatever it is you want to solve, and then **ENTER**.

The T/S 1ØØØ can not only add, but also subtract, multiply (using a star * instead of the usual times sign — this is fairly common on computers) and divide (using / instead of ÷). Try these out.

+, −, * and / are *operations*, and the numbers they operate on are their *operands*.

The computer can also raise one number to the power of another by using the operation ** (shifted H): type

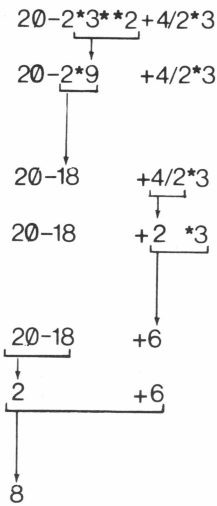
PRINT 2**3

and you will get the answer 8 (2^3, or 2 cubed).

The T/S 1ØØØ will also compute combinations of the operations. For example,

PRINT 2Ø−2*3**2+4/2*3

gives the answer 8. It does it in a roundabout way, because first it works out all the powers (**) in order from left to right; then all the multiplications and divisions (* and /), again from left to right; and then the additions and subtractions (+ and −), yet again from left to right. Thus our example is worked out in the following stages:



first the powers

then the multiplications and divisions

and then the additions and subtractions

We formalize this by giving each operation a *priority*, a number between 1 and 16. The operations with highest priority are evaluated first, and operations with equal priority are evaluated in order from left to right.

**	has priority 10
* and /	have priority 8
+ and −	have priority 6

When − is used to negate something, as when you write −1, it has priority 9. This is *unary* minus, as opposed to the *binary* minus in 3−1. (A unary operation has one operand, while a binary operation has two.) Note that on the T/S 1000 you cannot use + as a unary operation.

This order is absolutely rigid, but you can circumvent it by using parentheses: anything in parentheses is evaluated first and then treated as a single number, so that

PRINT 3*2+2

gives the answer 6+2=8, but

PRINT 3*(2+2)

gives the answer 3*4=12.

A combination like this is called an *expression* — in this case, an *arithmetic* or *numeric* expression, because the answer is a number. In general, whenever the computer is expecting a number from you, you can give it an expression instead and it will work out the answer.

You can write numbers with decimal points, and you can also use scientific notation, as is quite common on pocket calculators. In this, after an ordinary number (with or without a decimal point), you can write an *exponent part* consisting of the letter E, then maybe + or −, and then a number without a decimal point. The E here means "* 1Ø**," ("times 1Ø to the power of") so that

```
2.34EØ = 2.34 * 1Ø**Ø = 2.34
2.34E3 = 2.34 * 1Ø**3 = 234Ø
2.34E−2 = 2.34 * 1Ø**−2 = Ø.Ø234      and so on.
```

(Try printing these out on the T/S 1ØØØ.)

It will help to understand this if you imagine the exponent part shifting the decimal point along to the right (for a positive exponent) or to the left (for a negative exponent).

You can also print more than one entry at a time, separating them with either commas (,) or semicolons (; on shifted X). If you use a comma, the next number will be displayed starting either as the left-hand margin or in the middle of the line in the 16th column. If you use a semicolon, the next number will be displayed immediately following the last one. Try

```
PRINT 1;2;3;4;5;6;7;8;9;1Ø
PRINT 1,2,3,4,5,6,7,8,9,1Ø
```

to see the differences. You can mix commas and semicolons within a single **PRINT** statement.

Summary

Functions: +, −, *, /, **
Expressions, scientific notations

Exercises

1. Try

```
PRINT 2.34EØ
PRINT 2.34E1
PRINT 2.34E2
```

and so on up to

```
PRINT 2.34E15
```

You will see that after a while the T/S 1ØØØ also starts using scientific notation. This is because it never uses more than 14 spaces to write a number. Similarly, try

PRINT 2.34E-1
PRINT 2.34E-2

and so on.

2. Try

PRINT 1,,2,,3,,,4,,,,5

A comma always moves you on toward the next number. Now try

PRINT 1;;2;;3;;;4;;;;5

Why is a string of semicolons no different from a single one?

3. **PRINT** gives only 8 significant digits. Try

PRINT 4294967295,4294967295 −429E7

This proves that the computer can hold all the digits of 4294967295, even though it is not prepared to display them all at once.

4. If you have some log tables, test out this rule:
Raising 10 to the power of a number is the same as taking the antilog of that number.
For example, type

PRINT 10**0.3010

and look up the antilog of 0.3010. Why are the answers not exactly equal?

5. The T/S 1000 uses *floating point* arithmetic, which means that it keeps separate the digits of a number (its *mantissa*) and the position of the point (the *exponent*). The answer is not always exact, even for whole numbers.

Type

PRINT 1E10+1−1E10,1E10−1E10+1

Numbers are held to about 9 1/2 digits' accuracy, so 1E10 is too big to be held exactly right. The inaccuracy (actually about 2) is more than 1, so the numbers 1E10 and 1E10+1 appear to the computer to be equal.
For an even more peculiar example, type

PRINT 5E9+1−5E9

Here the inaccuracy in 5E9 is only about 1, and the 1 to be added on, in fact, gets *rounded up* to 2. Here the numbers 5E9+1 and 5E9+2 appear to the computer to be equal.
The largest integer that can be held completely accurately is $2^{32}-1$ (4,294,967,295). The T/S 1000 rounds this to 8 significant digits and it is displayed as 4,294,967,300.

Chapter 12

Advanced Printing Techniques

You will recall that a **PRINT** statement has a list of items, each one an expression (or possibly nothing at all), and that they are separated by commas or semicolons. There are two more **PRINT** items used to tell the computer not *what*, but *where* to print. For example, **PRINT AT** 11, 16;"*" prints a star in the middle of the screen.

AT line, column

moves the **PRINT** position (the place where the next item is to be printed) to the line and column specified. Lines are numbered from Ø (at the top) to 21, and columns from Ø (on the left) to 31.

TAB column

moves the **PRINT** position to the column specified. It stays on the same line, or, if this would involve back-spacing, moves on to the next one. Note that the computer reduces the column number modulo 32 (it divides by 32 and takes the remainder); so **TAB** 33 means the same as **TAB** 1. For example (to print out a Contents page heading):

```
PRINT TAB 12;"CONTENTS";AT 3,1;
"CHAPTER";TAB 24; "PAGE"
```

Some small points:

(i) You should normally use semicolons with these new items, as we have done above. You can use commas (or nothing, at the end of the statement), but this means that after having carefully set up the **PRINT** position you immediately move it to either the left edge or the center of the screen.

(ii) Although **AT** and **TAB** are not functions, you have to type the function key (shifted **ENTER**) to get them.

(iii) You cannot print on the two bottom lines (22 and 23) of the screen. References to the "bottom line" usually mean line 21.

(iv) You can use **AT** to put the **PRINT** position even where there is already something printed; the old stuff will be overwritten.

There are two more statements connected with **PRINT**, namely **CLS** and **SCROLL**.

CLS clears the screen (but nothing else).

SCROLL moves the whole display up one line (losing the top line) and moves the **PRINT** position to the beginning of the bottom line.

To see how it works, run this program:

```
10 SCROLL
20 INPUT A$
30 PRINT A$
40 GOTO 10
```

Summary

PRINT items: **AT, TAB**
Statements: **CLS, SCROLL**

Exercises

1. Try running this:

```
10 FOR I=0 TO 20
20 PRINT TAB 8*I;I;
30 NEXT I
```

This shows what is meant by the **TAB** number's being reduced modulo 32. For a more interesting example, change the 8 in line 20 to a 6.

Chapter 13

The Character Set

The letters, digits, punctuation marks and so on that can appear in strings are called *characters*, and they make up the alphabet, or *character set*, that the T/S 1ØØØ uses. Most of these characters are single *symbols*, but there are some, called *tokens*, that represent whole words, such as **PRINT**, **STOP**, **, and so on.

There are 256 characters altogether, and each one has a code between Ø and 255. A complete list of them appears in the Appendix. To convert between codes and characters, there are two functions, **CODE** and **CHR$**.

CODE is applied to a string and gives the code of the first character in the string (or Ø if the string is empty).

CHR$ is applied to a number and gives the single character string whose code is that number.

This program prints out the entire character set.

```
10 LET A=0
20 PRINT CHR$ A;
30 LET A=A+1
40 IF A<256 THEN GOTO 20
```

At the top you can see the symbols ", £, $ and so on up to Z; all appear on the keyboard and can be typed in when you have the L cursor. Further on, you can see the same characters, but in white on black (inverse video); these are also obtainable from the keyboard. If you press **GRAPHICS** (shifted 9), the cursor will come up as G: this means *graphics mode*. If you type in a symbol, it will appear in its inverse video form, and this will go on until you press **GRAPHICS** again. **DELETE** will have its usual meaning. Be careful not to lose the G cursor among all the inverse video characters you've just typed in.

When you've experimented a bit, you should still have the character set at the top; if not, then run the program again. At the beginning are space and ten patterns of black, white and grey blobs; further on, there are eleven more. These are called the *graphics symbols* and are used for drawing pictures. You can enter these from the keyboard, again using graphics mode (except for space, which is an ordinary symbol using the L cursor; the black square is inverse space). Use the 20 keys that have graphics symbols written on them. For instance, suppose you want the symbol ▚, which is the T key. Press **GRAPHICS** to get the G cursor; and then press shifted T. From the previous description of the graphics mode, you would expect to get an inverse video symbol; but shifted T is normally <>, a token, and tokens have no inverses: so you get the graphics symbol ▚ instead.

Here are the 22 graphic symbols.

Symbol	*Code*	*How obtained*	*Symbol*	*Code*	*How obtained*
	0	K or L **SPACE**		128	G **SPACE**
	1	G shifted 1		129	G shifted Q
	2	G shifted 2		130	G shifted W
	3	G shifted 7		131	G shifted 6
	4	G shifted 4		132	G shifted R
	5	G shifted 5		133	G shifted 8
	6	G shifted T		134	G shifted Y
	7	G shifted E		135	G shifted 3
	8	G shifted A		136	G shifted H
	9	G shifted D		137	G shifted G
	10	G shifted S		138	G shifted F

Now look at the character set again. The tokens stand out quite clearly in two blocks: a small group of three (**RND**, **INKEY$** and **PI**) after Z, and a larger group (starting with the quote image after ▣, and running from **AT** up to **COPY**).

The rest of the characters all seem to be ?. This is actually just the way they get printed; the real question mark is between : and (. Of the others, some are for control characters like ◊, **EDIT** and **ENTER**, and the rest are for characters that have no special meaning for the T/S 1ØØØ at all.

Summary

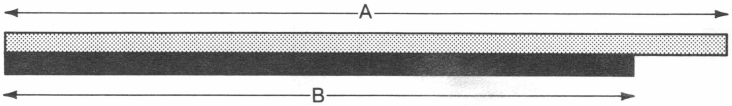
Functions: **CODE, CHR$**

Exercises

1. Imagine the space for one symbol divided up into four quarters: ⊞. If each quarter can be either black or white, there are 2*2*2*2 = 16 possibilities. Find them all in the character set.

2. Imagine the space for one symbol divided into two horizontally: ⊟. If each half can be black, white or gray, there are 3*3 = 9 possibilities. Find them all.

3. The characters in exercise 2 are designed to be used in horizontal bar charts, using two colors, gray and black. Write a program that inputs two numbers A and B (both between Ø and 32) and draws a bar chart for them



You will need to start off by printing "▓"; then change to either "▒" or "▄", depending on whether A is more or less than B.

What does your program do if A and B are not whole numbers? Or if they are not in the range Ø to 32? A good — "user friendly" is the fashionable term — program will do something sensible and useful.

4. There are two different all-gray characters on the keyboard, on A and H. If you look at them closely, you will see that the one on H is like a miniature chessboard, while the one on A is like a sideways chessboard. If you print them next to each other, you will see that they don't join up properly. The one on A is used to join up neatly with ▒ and ▒ (on S and D), whereas the one on H joins up neatly with ▓ and ▓ (on F and G).

5. Run this program:

```
10 INPUT A
20 PRINT CHR$ A;
30 GOTO 10
```

If you will experiment with it, you will find that for **CHR$**, A is *rounded* to the nearest whole number; and if A is not in the range Ø to 255, the program stops with report B, "integer out of range."

6. Using the codes for the characters, we can extend the concept of "alphabetical ordering" to cover strings containing any characters, not just letters. If, instead of thinking in terms of the usual alphabet of 26 letters, we use the extended alphabet of 256 characters in the same order as their codes, the principle is exactly the same. For instance, these strings are in alphabetical order:

" ZACHARY"
" ■ "
"(ASIDE)"
"123 TAXI SERVICE"
"AASVOGEL"
"AARDVARK "
"ZACHARY"
" AARDVARK"

Here is the rule. First, compare the first characters in the two strings. If they differ, the code of one of them is less than that of the other, and the string of which it is the first character is the earlier (lesser) of the two strings. If they are the same, then go on to compare the next characters. If in this process one of the strings runs out before the other, that string is the earlier; otherwise they are obviously equal.

Type in again the program in exercise 5 of Chapter 7 (the one that inputs two strings and prints them in order) and use it to experiment.

7. This program prints a screenful of random black and white graphics characters:

```
10 LET A=INT (16*RND)
20 IF A>=8 THEN LET A=A+120
30 PRINT CHR$ A;
40 GOTO 10
```

(How?)

Chapter 14

Graphics

Here are some of the more attractive features of the T/S 1ØØØ; they utilize *pixels* (picture elements). The screen you use for display has 22 lines and 32 columns, making 22*32 = 7Ø4 character positions, each containing 4 pixels.

A pixel is specified by two numbers, its *coordinates*. The first, its *x-coordinate*, denotes how far it is across from the extreme left-hand column (remember, X is ACROSS); and the second, its *y-coordinate*, tells how far up it is from the bottom. These coordinates are usually written as a pair in parenthesis, so (Ø,Ø), (63, Ø), (Ø, 43) and (63, 43) are the bottom left-, bottom right-, top left- and top right-hand corners.

The statement

PLOT x-coordinate, y-coordinate

blacks in the pixel with these coordinates, while the statement

UNPLOT x-coordinate, y-coordinate

blanks it out.

Try this simple program

```
10 PLOT INT (RND*64), INT (RND*44)
20 INPUT A$
30 GOTO 10
```

This plots a random point each time you press **ENTER**.

Here is a more useful program. It plots a graph of the function **SIN** (a sine wave) for values between Ø and 2 pi radians.

```
10 FOR N=Ø TO 63
20 PLOT N,22+2Ø*SIN (N/32*PI)
30 NEXT N
```

This next one plots a graph of **SQR** (part of a parabola) between Ø and 4:

```
10 FOR N=Ø TO 63
20 PLOT N, 2Ø*SQR (N/16)
30 NEXT N
```

Notice that pixel coordinates are different from the line and column in an **AT** item. You may find the diagram at the end of this chapter useful in working out pixel coordinates and line and column numbers.

Exercises

1. There are three differences among the numbers in an **AT** item and pixel coordinates; what are they?

Suppose a **PRINT** position corresponds to **AT** L, C (for line and column). Prove to yourself that the four pixels in that position have x-coordinates 2*C or 2*C+1, and y-coordinates 2*(21−L) or 2*(21−L)+1. (Look at the diagram.)

2. Alter the simple program so that it first fills the screen with black (a black square is an inverse video space), and then unplots random points. If you have only 1K of memory — the standard machine without extra memory — you will find yourself running out of space and will have to alter the program so that it uses only part of the screen.

3. Modify the **SIN** graph program so that before plotting the graph itself it prints a horizontal line of "−"s for an x-axis and a vertical line of "/"s for a y-axis.

4. Write programs to plot graphs of more functions: e.g., **COS**, **EXP**, **LN**, **ATN**, **INT** and so on. For eah one, you must make sure that the graph fits the screen, so you will need to consider

(i) over what range you are going to take the functions (corresponding to the range 0 to 2 pi radians for the **SIN** graph).

(ii) where on the screen to put the x-axis (corresponding to 22 in line 20 in the **SIN** graph program).
(iii) how to scale the y-axis of the graph (corresponding to 20 in line 20 of the **SIN** graph program).

You will find that **COS** is the easiest — it's just like **SIN**.

5. Run this:

```
10 PLOT 21,21
20 PRINT "HEAVY QUOTES"
30 PLOT 46,21
```

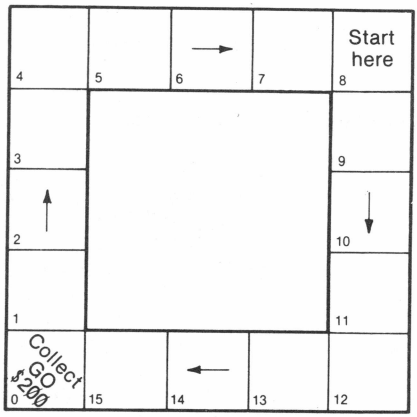
PLOT moves the **PRINT** position to the first space after the **PLOT** location. (**UNPLOT** does too.)

6. This subroutine draws a fairly straight line from the pixel (A,B) to the pixel (C,D). Use it as part of some main program that supplies the values A, B, C, D.

(If you do not have a memory expansion board then you will probably need to omit the **REM** statements.)

```
1000 LET U=C-A
1005 REM U SHOWS HOW MANY STEPS OVER
WE NEED TO GO
1010 LET V=D-B
1015 REM V SHOWS HOW MANY STEPS UP
1020 LET D1X=SGN U
1030 LET D1Y=SGN V
1035 REM (D1X,D1Y) IS A SINGLE STEP IN A DIAGONAL
DIRECTION
1040 LET D2X=SGN U
1050 LET D2Y=0
1055 REM (D2X,D2Y) IS A SINGLE STEP LEFT OR RIGHT
1060 LET M=ABS U
1070 LET N=ABS V
1080 IF M>N THEN GOTO 1230
1090 LET D2X=0
1100 LET D2Y=SGN V
1105 REM NOW (D2X,D2Y) IS A SINGLE STEP UP OR DOWN
1110 LET M=ABS V
1120 LET N=ABS U
1130 REM M IS THE LARGER OF ABS U AND ABS V, N IS THE
SMALLER
1140 LET S=INT (M/2)
1145 REM WE WANT TO MOVE FROM (A,B) TO (C,D) IN M
STEPS USING N UP-DOWN OR RIGHT-LEFT STEPS D2, AND
M-N DIAGONAL STEPS D1, DISTRIBUTED AS EVENLY AS
POSSIBLE
```

```
1150 FOR N=Ø TO M
1160 PLOT A,B
1170 LET S=S+N
1180 IF S<M THEN GOTO 1230
1190 LET S=S-M
1200 LET A=A+D1X
1210 LET B=B+D1Y
1215 REM A DIAGONAL STEP
1220 GOTO 1250
1230 LET A=A+D2X
1240 LET B=B+D2Y
1245 REM AN UP-DOWN OR RIGHT-LEFT STEP
1250 NEXT N
1260 RETURN
```



The last part (lines 1150 on) mixes the M–N steps D1 evenly with the N steps D2. Imagine a Monopoly board with M squares around the edge, numbered from Ø to M–1. The square you are on at any time is number S, starting at the corner opposite GO. Each move takes you N squares around the board, and in the straight line on the screen you make either a left-right / up-down step (if you pass GO on the board), or a diagonal step otherwise. Since your total journey on the board is M*N steps, or all the way around N times, you pass GO N times, and evenly spaced out in your M steps are N left-right / up-down steps.

Adjust the program so that if another parameter, E, is 1, the line is drawn in black (as here); and if it is Ø, the line is drawn in white (using **UNPLOT**). You can then delete a line you've just drawn by undrawing it.

An example: this is the pixel (57,32)
Columns
Lines
Pixel y-coordinates
Pixel x-coordinates
You cannot PRINT or PLOT on the bottom two lines.

Chapter 15

Functions

Mathematically, a function is a rule for giving a number (the *result*) in exchange for another (the *argument*, or *operand*) and so is really a unary operation. The T/S 1ØØØ has some of these built into it; their names are the words written under the keys. **SQR**, for instance, is the familiar square root function, and

PRINT SQR 9

gives 3, the square root of 9. (To get **SQR**, you first press the **FUNCTION** key — shifted **ENTER**. This changes the cursor to F. Now press the **SQR** key (H): SQR appears on the screen, and the cursor changes back to L. The same method works for all the words that are written underneath the keys, nearly all of which are function names.)

Try

PRINT SQR 2

You can test the accuracy of the answer by

PRINT SQR 2***SQR** 2

which ought to give 2. Note that both **SQR**s are worked out before the *, and in fact all functions (except one — **NOT**) are worked out before the five operations +, −, *, / and **. Again, you can circumvent this rule using parentheses:

PRINT SQR (2*2)

gives 2.

Here are some more functions (There is a complete list in Chapter 21). If your math is not up to understanding some of these, it doesn't matter — you will still be able to use the computer.

SGN The sign function (sometimes called *signum* to avoid confusion with **SIN**). The result is −1, Ø or +1 according to whether the argument is negative, zero or positive

ABS The absolute value, or modulus. The result is the argument made positive, so that

ABS −3.2 = **ABS** 3.2 = 3.2

SIN, **COS**, **TAN**, **ASN** arcsin, **ACS** arccos, **ATN** arctan — The trigonometric functions. These work in radians, not degrees.

LN natural logarithm (to base 2.718281828..., alias e)

EXP exponential function

SQR square root

INT integer part. This always rounds down, so **INT** 3.9 = 3 and **INT** −3.9 = −4. (An *integer* is a whole number, possibly negative.)

PI π = 3.14159265 ..., the circumference in inches of a circle one inch across. **PI** has no argument. (Type function π under the M key.)

RND Neither has **RND** an argument. It yields a random number between Ø (which value it can take) and 1 (which it cannot).

All these except **PI** and **RND** are unary operations with priority 11. (**PI** and **RND** are *nullary* operations, because they have no operands.)

The trigonometric functions, as well as **EXP**, **LN** and **SQR**, are generally calculated to 8 digits accuracy.

RND and **RAND**: These are both on the same key, but whereas **RND** is a function, **RAND** is a keyword, like **PRINT**. **RAND** is used for controlling the randomness of **RND**.

RND is not truly random, but instead follows a fixed sequence of 65536 numbers that happen to be so jumbled up as to appear random (**RND** is

pseudo-random). You can use **RAND** to start **RND** off at a definite place in this sequence by typing **RAND**, then a number between 1 and 65535, and then **ENTER**. It's not so important to know where a given number starts **RND** off, as that the same number after **RAND** will always start **RND** off at the same place. For instance, type

RAND 1 (and **ENTER**)

and then

PRINT RND

and type both these in turn several times. (Remember to use **FUNCTION** to get **RND**.) The answer from RND will always be Ø.ØØ22735596, not a very random sequence

RAND Ø

(or you can leave out the Ø) acts slightly differently: it determines where to start **RND** off by how long the television has been on, and this should be genuinely random.

Note: In some dialects of BASIC you must always enclose the argument of a function in brackets. This is not the case in Sinclair BASIC.

Summary:

Statement: **RAND**
Functions: **SGN, ABS, SIN, COS, TAN, ASN, ACS, ATN, LN, EXP, SQR, INT, PI, RND**

Exercises

1. To get common logarithms (to base 1Ø), which are what you'd look up in log tables, divide the natural logarithm by **LN** 1Ø. For example, to find log 2,

PRINT LN 2/**LN** 1Ø

which gives the answer Ø.3Ø1Ø3.

Try doing multiplication and division using logs, using the T/S 1ØØØ as a set of log tables in this way. Check the answers using * and /. The direct way is more accurate.

2. **EXP** and **LN** are *inverse* functions in the sense that if you apply one and then the other, you get back to your original number. For instance,

LN EXP 2 = **EXP LN** 2 = 2

The same also holds for **SIN** and **ASN**, for **COS** and **ACS**, and for **TAN** and **ATN**. You can use this to test how accurately the computer works out these functions.

3. π radians are 180°. To convert from degrees to radians, you divide by 180 and multiply by π : thus

PRINT TAN (45/180***PI**)

gives tan 45° (1).
To get from radians to degrees, divide by π and multiply by 180.

4. Try

PRINT RND

a few times to see how the answer varies. Can you detect any pattern? (Unlikely.)
How would you use **RND** and **INT** to get a random whole number between 1 and 6, to represent the throw of a die? (Answer: **INT** (**RND** *6) + 1.)

5. Test this rule:
Suppose you choose a number between 1 and 872 and type

RAND and then your number (and **ENTER**)

The next value of **RND** will be

(75 * (your number + 1) − 1) / 65536

6. (For mathematicians only.)
Let p be a (large) prime, and let a be a primitive root modulo p.
Then if b_i is the residue of a^i modulo p ($1 <= b_i <= p-1$), the sequence

$$\frac{b_i - 1}{p - 1}$$

is a cyclical sequence of p − 1 distinct numbers in the range 0 to 1 (excluding 1). By choosing a suitably, you can make these look fairly random.
65537 is a Mersenne prime, $2^{16}-1$. Use this, along with Gauss's law of quadratic reciprocity, to show that 75 is a primitive root modulo 65537.
The T/S 1000 uses p=65537 and a=75, and stores some arbitrary b_i-1 in memory. The function **RND** involves replacing b_i-1 in memory by $b_{i+1}-1$, and yielding the result $(b_{i+1}-1)/(p-1)$. **RAND** n (with $1 <= n <= 65535$) makes b_i equal to n+1.

7. **INT** always rounds down. To round to the nearest integer, add 0.5 first. For instance,

INT (2.9 + Ø.5) = 3 **INT** (2.4 + Ø.5) = 2
INT (−2.9 + Ø.5) = −3 **INT** (−2.4 + Ø.5) = −2

Compare these with the answers you get when you don't add Ø.5.

8. Try (type the symbol π ; the screen will show **PI**)

PRINT PI, PI−3, PI−3.1, PI−3.14, PI−3.141

This shows how accurately the computer stores π .

Chapter 16

Time and Motion

Sometimes you will want to make the program take a specified length of time, and for this purpose you will find the **PAUSE** statement useful:

PAUSE n

stops computing for n frames of the television (at 60 frames per second). n can be up to 32767, which gives you just under 11 minutes; if n is any bigger, it means '**PAUSE** forever.'

You can always cut a pause short by pressing a key (note that a space, or £, will cause a break as well). You have to press the key down after the pause has started.

At the end of the pause, the screen will flash.

This program works the second hand (here just a single dot on the edge) of a clock.

```
  5 REM FIRST WE DRAW THE CLOCK FACE
 10 FOR N=1 TO 12
 20 PRINT AT 10-10*COS (N/6*PI), 10+10*SIN (N/6*PI); N
 30 NEXT N
 35 REM NOW WE START THE CLOCK
 40 FOR T=0 TO 10000
 45 REM T IS THE TIME IN SECONDS
 50 LET A=T/30*PI
 60 LET SX=21+18*SIN A
 70 LET SY=22+18*COS A
200 PLOT SX,SY
300 PAUSE 42
310 UNPLOT SX,SY
400 NEXT T
```

This clock will run down after about 2 3/4 hours because of line 40, but you can easily make it run longer. Note how the timing is controlled by line 300. You might expect **PAUSE** 60 to make it tick once a second, but the computing takes time as well and has to be allowed for. This is best done by trial and error, timing the computer clock against a real one and adjusting line 300 until they agree. (You can't do this very accurately; an adjustment of one frame in one second is 2% or half an hour in a day).

The function **INKEY$** (which has no argument) reads the keyboard. If you are pressing exactly one key, the result is the character which that key gives in **L** mode; otherwise the result is the empty string. The control characters do not have their usual effect, but give results like **CHR$** 118 for **ENTER** — they are printed as '?'.

Try this program, which works like a typewriter.

```
10 IF INKEY$ <> "" THEN GOTO 10
20 IF INKEY$ = "" THEN GOTO 20
30 PRINT INKEY$
40 GOTO 10
```

Here line 10 waits for you to lift your finger off the keyboard and line 20 waits for you to press a new key.

Remember that unlike **INPUT**, **INKEY$** doesn't wait for you. So don't type **ENTER**; on the other hand, if you don't type anything at all, you have missed your chance. In this program, line 10 "waits for you" by the **GOTO** 10 repetition if no key is pressed.

Exercises

1. What happens if you leave out line 10 in the typewriter program?

2. Why can't you type space or £ in the typewriter program?

Here is a modified program that gives you a space if you type cursor right (shifted 8).

```
10 IF INKEY$<>"" THEN GOTO 10
20 IF INKEY$="" THEN GOTO 20
30 LET A$=INKEY$
40 IF A$=CHR$ 115 THEN GOTO 110
90 PRINT A$
100 GOTO 10
110 PRINT " ";
120 GOTO 10
```

Note that we read **INKEY$** into A$ in line 30. It would be possible to omit this and replace A$ by **INKEY$** in lines 40 and 90, but there would always be a chance that **INKEY$** could change between the lines.

Add some more program so that if you type **ENTER** (**CHR$** 118) it gives you a new line.

3. You can also use **INKEY$** in conjunction with **PAUSE**, as in this alternative typewriter program.

```
10 PAUSE 40000
20 PRINT INKEY$
30 GOTO 10
```

To make this work, why is it essential that a **PAUSE** not finish if it finds you already pressing a key when it starts?

This method has the disadvantage that the screen flashes, but in fast mode it is the only way of doing it. When you run a program in fast mode, notice that the computer uses the pause to display the television picture.

4. The following program makes the computer display a number, which you (or an innocent victim) must type in reponse. To begin with, you have a second to do it in, but if you get it wrong, you get a longer time for the next number; whereas if you get it right, you get less time for the next one. The idea is to get it going as fast as possible, and then press Q to find your score — the higher the better.

```
10 LET T=60
15 REM T=NUMBER OF FRAMES PER TURN —
   INITIALLY 60 FOR 1 SECOND
20 LET A$=CHR$ INT (RND*10+CODE "0")
30 REM A$ IS A RANDOM DIGIT
40 PRINT A$
50 PAUSE T
60 LET B$=INKEY$
70 IF B$="Q" THEN GOTO 200
80 IF A$=B$ THEN GOTO 150
90 PRINT "NO GOOD"
100 LET T=T*1.1
```

```
110 GOTO 20
150 PRINT "OK"
160 LET T=T*0.9
170 GOTO 20
200 SCROLL
210 PRINT "YOUR SCORE IS ";INT (500/T)
```

5. (For fun.) Try this:

```
10 IF INKEY$ = "" THEN GOTO 10
20 PRINT AT 11,8; "KEYBOARD TOUCHED"
30 IF INKEY$ <> "" THEN GOTO 30
40 PRINT AT 11,14; ' '
50 GOTO 10
```

Chapter 17

Arrays

An array is a set of variables, or *elements*, all with the same name and distinguished only by a number (the *subscript*) written in parentheses after the name. For example,the name could be A (like control variables of **FOR - NEXT** loops, the name of an array must be a single letter), and twelve variables would then be A(1), A(2), and so on up to A(12).

The elements of an array are called *subscripted* variables, as opposed to the *simple* variable you are already familiar with.

Before you can use an array you must reserve some space for it inside the computer. To do this you use a **DIM** (for dimension) statement.

DIM A(12)

sets up an array called A with *dimension* 12 (i.e., there are 12 subscripted variables A(1), ..., A(12)), and initializes the 12 values to Ø. It also deletes any array called A that existed previously. (But not a simple variable. An array and a simple numerical variable with the same name can coexist, and there shouldn't be any confusion between them, because e array variable always has a subscript.)

The subscript can be an arbitrary numerical expression, so now you can write

```
10 FOR N=1 TO 12
20 PRINT A(N)
30 NEXT N
```

You can also set up arrays with more than one dimension. In a two-dimensional array you need two numbers to specify one of the elements — rather like the line and column numbers to specify a character position on the television screen — so it has the form of a table. Alternatively, if you imagine the line and column numbers (two dimensions) as referring to a printed page, you could have an extra dimension for the page numbers. Of course, we are talking about numeric arrays; the elements would not be printed characters as in a book, but numbers. Think of the elements of a three-dimensional array C as being specified by C (page number, line number, column number).

For example, to set up a two-dimensional array B with dimension 3 and 6, you use a **DIM** statement

```
DIM B(3,6)
```

This then gives you 3*6 = 18 subscripted variables

```
B(1,1),  B(1,2), ... ,  B(1,6)
B(2,1),  B(2,2), ... ,  B(2,6)
B(3,1),  B(3,2), ... ,  B(3,6)
```

The same principle works for any number of dimensions.

Although you can have a number and an array with the same name, you cannot have two arrays with the same name, even if they have different numbers of dimensions.

There are also string arrays. The strings in an array differ from simple strings in that they are of fixed length, and assignment to them is always Procrustean. Another way of thinking of them is as arrays (with one extra dimension) of single characters. The name of the string array is a single letter followed by $, and a string array and a simple string variable cannot have the same name (unlike the case for numbers).

Suppose, then, that you want an array A$ of five strings. You must decide how long these strings are to be — let us suppose that 10 characters each is long enough. You then say

```
DIM A$(5,10)          (type this in)
```

This sets up a 5 * 1Ø array of characters, but you can also think of each row as being a string:

A$(1) =	A$(1,1)	A$(1,2)	...	A$(1,1Ø)
A$(2) =	A$(2,1)	A$(2,2)	...	A$(2,1Ø)
: :	:	:	:	:
A$(5) =	A$(5,1)	A$(5,2)	...	A$(5,1Ø)

If you give the same number of subscripts (two in this case) as there were dimensions in the **DIM** statement, you get a single character; but if you omit the last one, you get a fixed-length string. So, for instance, A$(2,7) is the 7th character in the string A$(2); using the slicing notation, we could also write this as A$(2)(7). Now type

```
LET A$(2)="1234567890"
```

and

```
PRINT A$(2),A$(2,7)
```

You get

```
1234567890        7
```

For the last subscript (the one you can omit), you can also have a slice so that, for example,

```
A$(2,4 TO 8) = A$(2)(4 TO 8) = "45678"
```

Remember:

In a string array, all the strings have the same, fixed length.

The **DIM** statement has an extra number (the last one) to specify this length.

When you write down a subscripted variable for a string array, you can put in an extra number, or a slicer, to correspond with the extra number in the **DIM** statement.

Summary

Arrays (the way the T/S 1ØØØ handles string arrays is slightly nonstandard).

Statements: **DIM**

Exercises

1. Set up an array M$ of twelve strings in which M$(I) is the name of the month.

2. (Hint: the **DIM** statement will be **DIM** M$(12,9).) Test it by printing out all the M$(I) (use a loop). Type

```
PRINT "NOW IS THE MONTH OF";M$(5);"ING"; "WHEN TINY
TOTS ARE PLAYING"
```

What can you do about all those spaces?

2. You can have string arrays with no dimensions. Type

```
DIM A$(10)
```

and you will find that A$ behaves just like a string variable, except that it always has length 10, and assignment to it is always Procrustean.

3. READ, DATA and RESTORE

Most BASICs (but not Sinclair BASIC) have three statements called READ, DATA and RESTORE.

A DATA statement is a list of expressions, and taking all the DATA statements in the program gives one long list of expressions, the *DATA list*.

READ statements are used to assign these expressions, one by one, to variables:

```
READ X
```

for instance, assigns the current expression in the DATA list to the variable X and moves on to the next expression for the next READ statement.

(RESTORE moves back to the beginning of the DATA list.)

In theory, you can always replace READ and DATA statements by LET statements; however, one of their major uses is for initializing arrays, as in this program:

```
 5 REM THIS PROGRAM WILL NOT WORK IN SINCLAIR BASIC
10 DIM M$(12,3)
20 FOR N=1 TO 12
30 READ M$(N)
40 NEXT N
50 DATA "JAN","FEB","MAR","APR"
60 DATA "MAY","JUN","JUL","AUG"
70 DATA "SEP","OCT","NOV","DEC"
```

If you want to run this program only once, you might well replace line 30 by

```
30 INPUT M$(N)
```

and you won't have any extra typing to do. However, if you want to save the program, you certainly won't want to type in the months everytime you run it.

We suggest that you use this method:

(i) Initialize the array using a **FOR-NEXT** loop and an **INPUT** statement as described above.

(ii) Delete the **FOR-NEXT** loop and the **INPUT** statement. (But not with **NEW**, because you want to preserve the array.)

(iii) Type in the rest of the program and save it. This will save the variables as well, including the array.

(iv) When you load the program back, you will also load the array.

(v) When you run the program, *do not use RUN*, which clears the variables. Use **GOTO** instead.

Chapter 18

Strings

One thing the T/S 1ØØØ can do that pocket calculators cannot do is deal with text. Type

PRINT "HI THERE. I AM YOUR T/S 1ØØØ." (" is shifted P.)

The greeting inside the quotes is called a *string* (meaning a string of characters) and can contain any characters you like except the string quote,". (But you can use the so-called *quote image*, "" (shifted Q), and this will be printed as ".)

A common typing error with strings is to leave out one of the quotes — this will give you the S marker.

If you are printing numbers out, you can use these strings to explain what the numbers mean. For instance, if you own a sportings good store, you might type

LET PRICEGOLF = 12.5Ø

and then

```
PRINT "THE PRICE OF GOLF BALLS IS ";
  PRICEGOLF;" PER DOZEN."
```

(Don't worry if this runs over into a second line.)

This statement displays three **PRINT** items: the string "THE PRICE OF GOLF BALLS IS," the number 12.5Ø (the value of the variable PRICEGOLF), and then the string "PER DOZEN." In fact, you can PRINT any number of items and any mixture of strings and numbers (or expressions). Note how the spaces in a string are just as much part of it as the letters. They are not ignored even at the end.

There are lots of things you can do with strings.

1. You can assign them to variables. However, the name of the variable must be special to show that its value is a string and not a number: it must be a single letter followed by $ (shifted U). For example, type

```
LET A$="BASKETBALL UNIFORM"
```

and

```
PRINT A$
```

2. You can add them together. This is often called *concatenation*, meaning "chaining together," and that is exactly what it does. Try

```
PRINT "JERSEY" + "AND SHORTS"
```

Note that you are missing a space. Now try

```
PRINT "JERSEY" + " AND SHORTS"
```

Notice the space before AND.

You cannot subtract, multiply or divide strings, or raise them to powers.

3. You can apply some functions to strings to get numbers, and vice versa.

LEN This is applied to a string, and the result is its length (the number of characters in the string). For instance

```
LEN "GLOVES" = 6
LEN "BATS" = 4
```

VAL This applied to a string, evaluates that string as an arithmetic expression. For instance (if A=9), **VAL** "1/2 + **SQR**A" = 3.5. If the string to which **VAL** is applied contains variables, then two rules must be obeyed.

(i) If the **VAL** function is part of a larger expression, it must be the first

item; e.g. 1Ø **LET** X = 7 + **VAL** "Y"must be changed to 1Ø **LET** X = **VAL** "Y" + 7.

(ii) **VAL** can only appear in the first coordinate of a **PRINT AT, PLOT** or **UNPLOT** statement
(see Chapters 12 and 14) e.g.

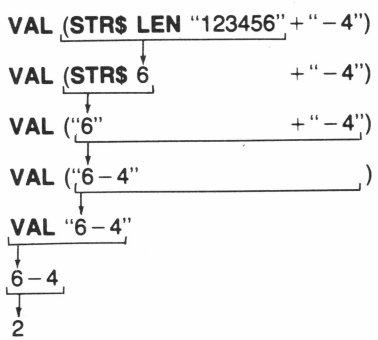
```
1Ø PLOT 5, VAL "X" must be changed to
1Ø LET Y = VAL "X"
15 PLOT 5, Y
```

STR$ When this is applied to a number, the result is what would appear on the screen if the number were displayed by a **PRINT** statement. For instance **STR$** 3.5 = "3.5".

4. Just as for numbers, you can combine these to make *string expressions*, like

```
VAL (STR$ LEN "123456" + "−4"
```

which is evaluated as



Summary

Operation: + (for strings)
Functions: **LEN, VAL, STR$**

Exercises

1. Type

```
LET A$="2+2"
```

and then

PRINT A$;" = ";**VAL** A$

Try changing A$ to more complicated items and doing the same; e.g.,

LEN A$="**ATN** 1*4"

(The answer here should be π.)

2. The string "" with no characters is called the *empty* or *null* string. It is the only string whose length is Ø. Remember that spaces are significant and an empty string is not the same as one containing spaces.

Do not confuse it with the quote image, "" (a single token, shifted Q). This is a special device to allow for the fact that you cannot write an ordinary string quote in the middle of a string (why not?). When the quote image appears in a string that has its quotes at the end (for instance, in the listing of a program), it shows up as two quote symbols, to distinguish it from the ordinary quote; but when it is displayed by a **PRINT** statement, it is as just one quote symbol.

Try

PRINT "X"; ""; "X"; """"; """"; ""; """"

3. Type

PRINT "2+2=";2+1

Chapter 19

Substrings

Given a string, a substring consists of a number of consecutive characters from it, taken in sequence. Thus "STRING" is a substring of "BIGGER STRING," but "B STING" and "BIG REG" are not.

There is a notation, called *slicing*, for describing substrings which can be applied to arbitrary string expressions. The general form is

string expression (start **TO** finish)

so that, for instance,

"ABCDEF" (2 **TO** 5) = "BCDE"

If you omit the start, then 1 is assumed; if you omit the finish, the length of the string is assumed. Thus

"ABCDEF" (**TO** 5) = "ABCDEF" (1 **TO** 5) = "ABCDE"
"ABCDEF" (2 **TO**) = "ABCDEF" (2 **TO** 6) = "BCDEF"
"ABCDEF" (**TO**) = "ABCDEF" (1 **TO** 6) = "ABCDEF"

(For what it's worth, you can also write this last one as "ABCDEF"().)
A slightly different form omits the **TO** and has just one number:

"ABCDEF" (3) = "ABCDEF" (3 **to** 3) = "C"

Although normally both start and finish must refer to existing parts of the string, this rule is overridden by another one: if the start is more than the finish, then the result is the empty string. So

"ABCDEF" (5 **to** 7)

gives error 3 (subscript error) because, since the string contains only 6 characters, 7 is too many, but

"ABCDEF" (8 **to** 7) = ""
and "ABCDEF" (1 **to** Ø) = ""

The start and finish must not be negative, or you get error B.

This next program makes B$ equal to A$, but omitting any trailing spaces.

```
1Ø INPUT A$
2Ø FOR N=LEN A$ TO 1 STEP -1
3Ø IF A$(N) 1Ø <> " " THEN GOTO 5Ø
4Ø NEXT N
5Ø LET B$=A$( TO N)
6Ø PRINT """";A$;"""","""";B$;"""
7Ø GOTO 1Ø
```

Note that if A$ is entirely spaces, then in line 5Ø we have N=Ø and A$ (**TO** N) = A$ (1 **TO** Ø) = "".

For string variables, we can not only extract substrings, but also assign substitute characters to them. For instance, type

```
1Ø LET A$="THAT IS FAR OUT"
and then 2Ø LET A$(5 TO 8)="******"
and 3Ø PRINT A$
```

Notice that since the substring A$ (5 **TO** 8) is only 4 characters long, only the first four stars have been used. This is a characteristic of assigning to substrings: the substring has to be exactly the same length afterwards as it was before. To make sure this happens, the string that is being assigned to it is cut off on the right if it is too long, or filled out with spaces if it is too short. This is called *Procrustean* assignment.

If you now change lines 2Ø and 3Ø to

```
2Ø LET A$()="TERRIFIC"
and 3Ø PRINT A$;"."
```

you will see that the same thing has happened again (this time with spaces put in) because A$() counts as a substring.

```
20 LET A$="TERRIFIC"
```

will place the period properly (but the string in line 10 has been replaced by the one in 20).

Slicing may be considered as having priority 12, so, for instance,

```
LEN "ABCDEF"(2 TO 5) = LEN ("ABCDEF"(2 TO 5)) = 4
```

Complicated string expressions will need parentheses around them before they can be sliced. For example,

```
"ABC"+"DEF"(1 TO 2)   = "ABCDE"
("ABC"+"DEF")(1 TO 2) = "AB"
```

Summary

Slicing, using **TO**. Note that this notation is used only on the T/S 1000.

Exercises

1. Some BASICs (*not* the Sinclair BASIC) have three functions, called LEFT$, RIGHT$ & MID$.

LEFT$(A$,N) gives the substring of A$ consisting of the first N characters.

RIGHT$(A$,N) gives the substring of A$ consisting of the characters from the Nth on.

MID$(A$,N1,N2) gives the substring of A$ consisting of N2 characters starting at the N1th.

How would you write these in Sinclair BASIC?

2. Try this sequence of commands:

```
10 LET A$="X*+*Y"
20 LET A$(2)=CHR$ 11
30 LET A$(4)=CHR$ 11
40 PRINT A$
```

A$ is now a string with string quotes inside it! So you can do this if you work hard enough, but if you had originally typed

```
10 LET A$="X"+"Y"
```

the part to the right of the equals sign would have been treated as an expression, giving A$ the value "XY."

Now type

```
50 LET B$="X""+""Y"
60 PRINT B$
```

You will find that although A$ and B$ look the same when printed out, they are not equal — try

```
70 PRINT A$=B$
```

The computer responds with "0" (the equation is false), because B$ contains mere quote image characters (with code 192), while A$ contains genuine string quote characters (with code 11).

3. Run this program:

```
 10 LET A$="LEN ""ABCD"""""
100 PRINT A$;"=";VAL A$
```

This will fail, because **VAL** does not treat the quote image "" as a string quote.

Insert some extra lines between 10 and 100 to replace the quote images in A$ by string quotes (which you must call **CHR$** 11), and try again.

4. Type in the subroutine that ignores trailing spaces, and write and run a program that uses it.

5. This subroutine deletes every occurrence of the string "SUPERMAN" from A$.

```
1000 FOR N=1 TO LEN A$-7
1020 IF A$(N TO N+7)="SUPERMAN" THEN LET A$(N TO
N+7)="********"
1030 NEXT N
1040 RETURN
```

Write a program that gives A$ various values (e.g., "SUPERMAN IS STRONG".) and applies the subroutine. Notice that "SUPERMAN" does not have to be the first word in A$.

Chapter 20

Sinclair BASIC Print Commands

This chapter covers the special BASIC statements needed to operate a printer with your T/S 1ØØØ.

The first two, **LPRINT** and **LLIST**, are just like **PRINT** and **LIST**, except that they use the printer instead of the television. (The L is a historical accident. When BASIC was invented, it generally used an electric typewriter instead of a television, so **PRINT** really did mean print. If you wanted masses of output, you would use a very fast line printer attached to the computer, and an **LPRINT** statement meaning "Line printer **PRINT**.")

Try this program, for example.

```
1Ø LPRINT "THIS PROGRAM",,,,
2Ø LLIST
3Ø LPRINT ,,"PRINTS OUT THE CHARACTER SET.",,,
4Ø FOR N=Ø TO 255
5Ø LPRINT CHR$ N;
6Ø NEXT N
```

The third statement, **COPY**, prints out a copy of the television screen. For

instance, get a listing on the screen of the program above, and type

COPY

You can always stop the printer when it is running by pressing the **BREAK** key (space).

If you execute these statements without the printer attached, you will usually lose the output and proceed to the next statement. However, sometimes the computer will get hung up, and you will need to use the **BREAK** key to rescue it.

Summary

Statements: **LPRINT, LLIST, COPY**

Note: None of these statements is standard BASIC, although **LPRINT** is used by some other computers.

Exercises

1. Try this:

```
10 FOR N=31 TO 0 STEP -1
20 PRINT AT 31-N,N;CHR$ (CODE "0" + N);
30 NEXT N
```

You will see a pattern of letters working down diagonally from the top right-hand corner until it reaches the bottom of the screen, when the program stops with error report 5.

Now change '**AT** 31−N,N' in line 20 to '**TAB** N'. The program will have exactly the same effect as before.

Now change **PRINT** in line 20 to **LPRINT**. This time there will be no error 5, which should not occur with the printer, and the pattern will continue an extra ten lines with the digits.

Now change '**TAB** N' to '**AT** 21−N,N' still using **LPRINT**. This time you will get just a single line of symbols. The reason for the difference is that the output from **LPRINT** is not printed immediately, but arranged in a *buffer* store, a picture one line long of what the computer will print

(i) when the buffer is full,
(ii) after an **LPRINT** statement that does not end in a comma or semicolon,
(iii) when a comma or **TAB** item requires a new line,
or (iv) at the end of a program, if there is anything left unprinted.

Number (iii) explains why our program with **TAB** works the way it does. As for **AT**, the line number is ignored and the **LPRINT** position (like the **PRINT** position, but for the printer instead of the television) is changed to the column

number. An **AT** item can never cause a line to be sent to the printer. (Actually, the line number after **AT** is not completely ignored; it has to be between −21 and +21 or an error will result. For this reason it is safest always to specify line Ø. The item '**AT** 21−N,N' in the last version of our program would be much better (although less illustrative) if replaced by '**AT** Ø,N'.)

2. Make a printed graph of **SIN** by running the program in Chapter 16 and then using **COPY**.

Chapter 21

The T/S 1000

for Those Who Understand BASIC

General

If you already know BASIC then you should not have much trouble using the T/S 1000; but it has one or two idiosyncrasies.

(i) Words are not spelled out, but have keys of their own — this is described in Chapter 3 (for keywords and shifted keys) and Chapter 15 (for function names). In the text, these words are printed in **BOLD TYPE**.

(ii) Sinclair BASIC lacks READ, DATA and RESTORE (but see exercise 3 of Chapter 17 concerning this), user-defined functions (FN and DEF; but **VAL** can sometimes be used), and multi-statement lines.

(iii) The string handling facilities are comprehensive but non-standard — see Chapters 18 and 19, and also Chapter 17 (for string arrays).

(iv) The T/S 1000 character set is completely its own.

(v) The television display is not, in general, memory-mapped.

(vi) If you are accustomed to using **PEEK** and **POKE** on a different machine, remember that all the addresses will be different on the T/S 1000.

Speed

The machine works at two speeds, called *compute and display* mode and *fast* mode.

In compute and display, the TV screen is generated continuously, and computing is done during the blank parts at the top and bottom of the picture.

In fast mode the television picture is turned off during computing and is displayed only at the end of a program, while waiting for **INPUT** data, or during a pause (see **PAUSE**).

Fast mode runs about four times as fast and should be used for programs with a lot of computing as opposed to output, or when typing in long programs.

Switching between speeds is done with the **FAST** and **SLOW** statements (q.v.).

The keyboard

T/S 1000 characters comprise not only the single *symbols* (letters, digits, etc.), but also the compound *tokens* (keywords, function names, etc.; these are printed here in **BOLD TYPE**), and all are entered from the keyboard rather than being spelled out. To fit this in, some keys have up to five distinct meanings, given partly by shifting the keys (i.e., pressing the **SHIFT** key at the same time as the required one) and partly by having the machine in different modes.

The mode is indicated by the *cursor*, an inverse video letter that shows where the next character from the keyboard will be inserted.

K mode (for keywords) occurs automatically when the machine is expecting a command or program line (rather than **INPUT** data), and from this position on the line it knows it should expect a line number or a keyword. This is at the beginning of the line, or just after some digits at the beginning of the line, or just after **THEN**. If unshifted, the next key will be interpreted as either a keyword (these are written *above* the keys) or a digit.

L mode (for letters) normally occurs at all other times. If unshifted, the next key will be interpreted as the main symbol on that key.

In both K and L modes, a shifted key will be interpreted as the subsidiary red character in the top right-hand corner of the key.

F mode (for functions) occurs after **FUNCTION** (shifted **ENTER**) is pressed and lasts for one key depression only. That key will be interpreted as a function name, which appears *under* the keys.

G mode for graphics occurs after **GRAPHICS** (shifted 9) is pressed and lasts until it is pressed again. An unshifted key will give the inverse video of its L mode interpretation. A shifted key will as well, provided that it is a symbol; but if the shifted key would normally give a token, in graphics mode it will give the graphics symbol that appears in the bottom right-hand corner of the key.

The screen

This has 24 lines, each 32 characters long, and is divided into two parts. The top part is at most 22 lines, and displays either a listing or program output. The bottom part, at least two lines, is used for inputting commands, program lines and **INPUT** data, and also for displaying reports.

Keyboard input: this appears in the bottom lines of the screen as it is

typed, each character (single symbol or compound token) being inserted just before the cursor. The cursor can be moved left with ◊ (shifted 5) or right with ◊ (shifted 8). The character before the cursor can be removed with **DELETE** (shifted Ø). (Note: the whole line can be deleted by typing **EDIT** (shifted 1) followed by **ENTER**.)

When **ENTER** is pressed, the line is executed, entered into the program, or used as **INPUT** data as appropriate, unless it contains a syntax error, in which case the symbol S appears just before the error.

As program lines are entered, a listing is displayed in the top half of the screen. The last line to be entered is called the *current line* and is indicated by the symbol ▶, but this can be changed by using the keys ◊ (shifted 6) and ◊ (shifted 7). If **EDIT** (shifted 1) is pressed, the current line is brought down to the bottom part of the screen and can be edited.

When a command is executed or a program run, output is displayed in the top half of the screen and remains until a program line is entered, or **ENTER** is pressed with an empty line, or ◊ or ◊ is pressed. In the bottom part appears a report of the form m/n, where m is a code showing what happened (see Report Codes), and n is the number of the last line executed — or Ø for a command. The report remains until a key is pressed (and indicates K mode).

In certain circumstances, the **SPACE** key acts as a **BREAK**, stopping the computer with report D. This is recognized

(i) at the end of a statement while a program is running,

(ii) while the computer is looking for a program on tape,

or (iii) while the computer is using the printer (or by accident trying to use it when it is not there).

The BASIC

Numbers are stored to an accuracy of 9 or 1Ø digits. The largest number you can get is about $1Ø^{38}$, and the smallest (positive) number is about $4 * 1Ø^{-39}$.

A number is stored in the T/S 1ØØØ in floating-point binary with one exponent e ($1 <= e <= 255$), and four mantissa bytes m ($1/2 <= m < 1$). This represents the number $m * 2^{e-128}$.

Since $1/2 <= m < 1$, the most significant bit of the mantissa m is always 1. Therefore in actual fact we can replace it with a bit to show the sign — Ø for positive numbers, 1 for negative.

Zero has a special representation in which all 5 bytes are Ø.

Numeric variables have names of arbitrary length, starting with a letter and continuing with letters and digits. Spaces are ignored.

Control variables of **FOR-NEXT** loops have names a single letter long.

Numeric arrays have names a single letter long, which may be the same as the name of a simple variable. They may have arbitrarily many dimensions of arbitrary size. Subscripts start at 1.

Strings are completely flexible in length. The name of a string consists of a single letter followed by $.

String arrays can have arbitrarily many dimensions of arbitrary size. The name is a single letter followed by $ and may not be the same as the name of a string. All the strings in a given array have the same fixed length, which is specified as an extra, final dimension in the **DIM** statement. Subscripts start at 1.

Slicing: Substrings of strings may be specified by using *slicers*. A slicer can be

(i) empty
or
(ii) numerical expression
or
(iii) optional numerical expression **TO** optional numerical expression and is used in expressing a substring either by
(a) string expression (slicer)
or by
(b) string array variable (subscript, ..., subscript, slicer)
which means the same as
string array variable (subscript, ..., subscript)(slicer)

In (a), suppose the string expression has the value s$.
If the slicer is empty, the result is s$ considered as a substring of itself.
If the slicer is a numerical expression with value m, the result is the mth character of s$ (a substring of length 1).
If the slicer has the form (iii), suppose the first numerical expression has the value m (the default value is 1), and the second, n (the default value is the length of s$).
If 1 <= m <= n <= the length of s$, the result is the substring of s$, starting with the mth character and ending with the nth.
If Ø <= n < m, the result is the empty string.
Otherwise, error 3 results.
Slicing is performed before functions or operations are evaluated, unless brackets dictate otherwise.
Substrings can be assigned to (see **LET**).
The argument of a function does not need brackets if it is a constant or a (possibly subscripted or sliced) variable.

Function	**Type of operand**	**Result**
	(x)	
ABS	number	Absolute magnitude.
ACS	number	Arccosine in radians. Error A if x not in the range −1 to +1.
AND	binary operation, right operand always a number.	

	Numeric left operand:	A **AND** B = { A if B ≠ Ø ; Ø if B = Ø
	String left operand:	A$ **AND** B = { A$ if B ≠ Ø ; "" if B = Ø
ASN	number	Arcsine in radians. Error A if x not in the range −1 to +1.
ATN	number	Arctangent in radians.
CHR$	number	The character whose code is x, rounded down to the nearest integer. Error B if x not in the range Ø to 255.
CODE	string	The code of the first character in x (or Ø if x is empty string).
COS	number (in radians)	Cosine
EXP	number	e^x.
INKEY$	none	Reads the keyboard. The result is the character representing (in L mode) the key pressed if there is exactly one, else the empty string.
INT	number	Integer part (always rounds down).
LEN	string	Length.
LN	number	Natural logarithm (to base e). Error A if x < = Ø.
NOT	number	Ø if x ≠ Ø, 1 if x = Ø. **NOT** has priority 4.
OR	binary operation, both operands numbers	A **OR** B = { 1 if B ≠ Ø. ; A if B = Ø. **OR** has priority 2.
PEEK	number	The value of the byte in memory whose address is x (rounded to the nearest integer). Error B if x not in the range Ø to 65535.

PI	none	π (3.14159265...)
RND	none	The next pseudo-random number y in a sequence generated by taking the powers of 75 modulo 65537, subtracting 1 and dividing by 65536. $\emptyset <= y < 1$.
SGN	number	Signum: the sign (-1, $\emptyset$ or $+1$) of x.
SIN	number (in radians)	Sine.
SQR	number	Square root. Error B if $x < \emptyset$.
STR$	number	The string of characters that would be displayed if x were printed.
TAN	number (in radians)	Tangent.
USR	number	Calls the machine code subroutine whose starting address is x. On return, the result is the contents of the bc register pair.
VAL	string	Evaluates x (without its bounding quotes) as a numerical expression. Error C if x contains a syntax error, or gives a string value. Other errors possible, depending on the expression.
–	number	Negation

The following are binary operations:

+	Addition (on numbers), or concatenation (on strings)	
–	Subtraction	
*	Multiplication	
/	Division	
**	Raising to a power. Error B if left operand is negative.	
=	Equals	Both operands must be of the same type. The result is a number, 1 if the comparison holds, and $\emptyset$ if it does not.
>	Greater than	
<	Less than	
<=	Less than or equal to	
>=	Greater than or equal to	
<>	Not equal to	

Functions and operations have the following priorities:

Operation	*Priority*
Subscribing and slicing	12
All functions except **NOT** and unary minus	11
**	10
Unary minus	9
*,/	8
+,– (binary –)	6
=,>,<,<=,>=,<>	5
NOT	4
AND	3
OR	2

Statements

In this list,

@	represents a single letter
v	represents a variable
x,y,z	represents numerical expressions
m,n	represents numerical expressions that are rounded to the nearest integer
e	represents an expression
f	represents a string-valued expression
s	represents a statement

Note that arbitrary expressions are allowed everywhere (except for the line number at the beginning of a statement).

All statements except **INPUT** can be used either as commands or in programs (although they may be more sensible in one than the other).

CLEAR Deletes all variables, freeing the space they occupied.

CLS (CLear Screen) Clears the display file. See Chapter 26 concerning the display file.

CONT Suppose a/b were the last report with a non-zero. Then **CONT** has the effect

GOTO b if a ≠ 9
GOTO b+1 if a = 9 (**STOP** statement)

COPY Sends a copy of the display to the printer, if attached; otherwise does nothing.
Report D if **BREAK** pressed.

DIM@ $(n_1,...,n_k)$

Deletes any array with the name @ and sets up an array of numbers with k dimensions $n_1,...,n_k$. Initializes all the values to Ø.
Error 4 occurs if there is no room to fit the array in. An array is undefined until it is dimensioned in a **DIM** statement.

DIM@$ $(n_1, ..., n_k)$

Deletes any array or string with the name @$ and sets up an array of characters with k dimensions $n_1, ..., n_k$. Initializes all the values to "". This can be considered as an array of strings of fixed length n_k, with $k-1$ dimensions $n_1, ..., n_{k-1}$.
Error 4 occurs if there is no room to fit the array in. An array is undefined until it is dimensioned in a **DIM** statement.

FAST

Starts fast mode, in which the display file is displayed only at the end of the program, while **INPUT** data is being typed in, or during a pause.

FOR@ =x **TO** y

FOR@ =x **TO** y **STEP** 1

FOR@ =x **TO** y **STEP** z

Deletes any simple variable and sets up a control variable with value x, limit y, step z, and looping address 1 more than the line number of the **FOR** statement (−1 if it is a command). Checks if the initial value is greater (if step > = Ø) or less (if step < Ø) than the limit, and if so, then skips to statement **NEXT@** at the beginning of a line. See **NEXT@**.
Error 4 occurs if there is no room for the control variable.

GOSUB n

Pushes the number of the **GOSUB** statement onto a stack; then as **GOTO** n.
Error 4 can occur if there are not enough **RETURN**s.

GOTO n

Jumps to line n (or, if there is none, to the first line after that).

IF x **THEN** s

If x is true (non-zero), then s is executed.
The form '**IF** x **THEN** line number' is not allowed.

INPUT v

Stops (with no special prompt) and waits for the user to type in an expression; the value of this is assigned to v. In fast mode, the display file is displayed. **INPUT** cannot be used as a command; error 8 occurs if you try.
If the first character in the **INPUT** line is **STOP**, the program stops with report D.

LET v=e

Assigns the value of e to the variable v.
LET cannot be omitted.
A simple variable is undefined until it is assigned to in a **LET** or **INPUT** statement.
If v is a subscripted string variable, or a sliced string variable (substring), then the assignment is *Procrustean*: the string value of e is either truncated or filled out with spaces to the right, to make it the same length as the variable v.

LIST

LIST Ø

LIST n

Lists the program on the TV screen, starting at line n, and makes n the current line.
Error 4 or 5 if the listing is too long to fit on the screen; **CONT** will do exactly the same again.

LLIST

LLIST Ø

LLIST n

Like **LIST**, but using the printer instead of the television.
Should do nothing if the printer is not attached.
Stops with Report D if **BREAK** is pressed.

LOAD f

Looks for a program called f on tape and loads it and its variables. If f = "", then loads the first program available.
If **BREAK** is pressed, then
(i) if no program has yet been read in from tape, stops with report D and old program;
(ii) if part of a program has been read in, then executes **NEW**.

LPRINT ...

Like **PRINT**, but using the printer instead of the television. A line of text is sent to the printer
(i) when printing spills over from one line to the next,
(ii) after an **LPRINT** statement that does not end in a comma or a semicolon,

(iii) when a comma or **TAB** item requires a new line, or
(iv) at the end of the program, if there is anything left unprinted.
In an **AT** item, only the column number has any affect; the line number is ignored. An **AT** item never sends a line of text to the printer.
There should be no effect if the printer is absent.
Stops with report D if **BREAK** is pressed.

NEW

Restarts the BASIC system, deleting program and variables and using the memory up to but not including the byte whose address is in the system variable RAMTOP (bytes 16388 and 16389).

NEXT @

(i) Finds the control variable @.
(ii) Adds its step to its value.
(iii) If the step >= 0 and the value > the limit; or if the step < 0 and the value < the limit, then jumps to the looping line.
Error 1 if there is no control variable @.

PAUSE n

Stops computing and displays the display file for n frames (at 50 frames per second) or until a key is pressed. 0 <= n <= 65535, else error B. If n >= 32767, then the pause is not timed, but lasts until a key is pressed.

PLOT m,n

Blacks in the pixel (|m|,|n|); moves the **PRINT** position to just after that pixel.
0 <= |m| <= 63, 0 <= |n| <= 43, else error B.

POKE m,n

Writes the value n to the byte in store with address m.
0 <= m <= 65535, −255 <= n <= 255, else error B.

PRINT ...

The '...' is a sequence of **PRINT** items, separated by commas or semicolons. They are written to the display file for display on the television. The position (line and column) where the next character is to be printed is called the **PRINT** position.
A **PRINT** item can be
(i) empty, i.e., nothing
(ii) a numerical expression

First, a minus sign is printed if the value is negative.
Now let x be the modulus of the value.

If $x <= 10^{-5}$ or $x >= 10^{13}$, then it is printed using scientific notation. The mantissa part has up to eight digits (with no trailing zeros), and the decimal point (absent if only one digit) is after the first. The exponent part is E, followed by + or −, followed by one or two digits.

Otherwise x is printed in ordinary decimal notation with up to eight significant digits, and no trailing zeros after the decimal point. A decimal point right at the beginning is always followed by a zero, so, for instance, .Ø3 and Ø.3 are printed as such.

Ø is printed as a single digit Ø.

(iii) a string expression.

The tokens in the string are expanded, possibly with a space before or after.

The quote image character prints as ".
Unused characters and control characters print as ?.

(iv) **AT** m,n

The **PRINT** position is changed to line m (counting from the top), column n (counting from the left). $Ø <= |m| <= 21$, $Ø <= |n| <= 31$, else error B. If $|m| = 22$ or 23, error 5.

(v) **TAB** n

n is reduced modulo 32. Then, the **PRINT** position is moved to column n, staying on the same line unless this would involve backspacing, in which case it moves on to the next line. $Ø <= n <= 255$, else error B.

A semicolon between two items leaves the **PRINT** position unchanged, so that the second item follows immediately after the first. A comma, on the other hand, moves the **PRINT** position on at least one place; and after that, as many as are necessary to leave it in column Ø or 16, moving to a new line if necessary.

At the end of the **PRINT** statement, if it does not end in a semicolon or comma, a new line is started.

Error 4 (out of memory) can occur with 3K or less of memory.
Error 5 means that the screen is filled.

In both cases, the cure is **CONT**, which will clear the screen and allow the program to continue.

RAND

RAND Ø

RAND n

Sets the system variable (called SEED) used to generate the next value of **RND**. If n ≠ Ø, the SEED is given the value n; if n = Ø, it is given the value of another system variable (called FRAMES) that counts the frames so far displayed on the television, and so should be fairly random.
Error B occurs if n is not in the range Ø to 65535.

REM ...

No effect. '...' can be any sequence of characters except **ENTER**.

RETURN

Pops a line number from the **GOSUB** stack and jumps to the line after it.
Error 7 occurs when there is no line number on the stack. There is some mistake in your program; **GOSUB**s are not properly balanced by **RETURN**s.

RUN

RUN Ø

RUN n

CLEAR, and then **GOTO** n.

SAVE f

Records the program and variables on tape and calls it f.
SAVE should not be used inside a **GOSUB** routine.
Error F occurs if f is the empty string, which is not allowed.

SCROLL

Scrolls the display file up one line, losing the top line and making an empty line at the bottom.
Note that the new line is genuinely empty with just an **ENTER** character and no spaces.

SLOW

Puts the computer into compute and display mode, in which the display file is displayed continuously and computing is done during the spaces at the top and bottom of the picture.

STOP — Stops the program with Report 9. **CONT** will resume with the following line.

UNPLOT m,n — Like **PLOT**, but blanks out a pixel instead of blacking it in.

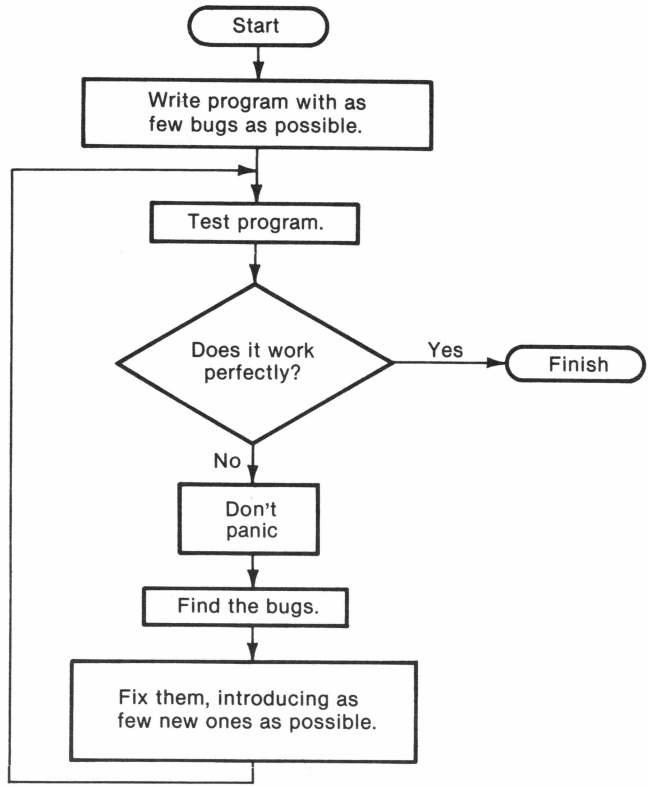
Chapter 22

Flowcharting and Debugging

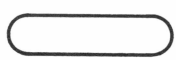
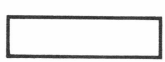
There is more to the art of programming computers than just knowing which statement does what. You will probably already have found that most of your programs have what are technically known as *bugs* when you first type them in: maybe just typing errors, or maybe mistakes in your own ideas of what the program should do. You might put this down to inexperience.

ALMOST EVERY PROGRAM STARTS OFF WITH BUGS IN IT

The general plan can be illustrated with a *flowchart*:



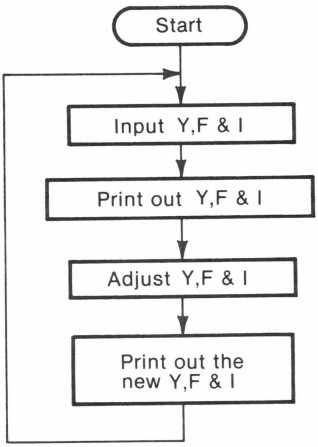
The idea is to follow the arrows from box to box, doing what each one tells you to do. We have used different sorts of boxes for different sorts of instructions:

A rounded box		is to start or finish.
A rectangular box		is a straightforward instruction.
A diamond		asks you to make some kind of decision before going on.

(These shapes are fairly widely used, but they are not mandatory.)

Flowcharts are often used for describing the large-scale structure of programs, with a subroutine in almost every box, so a flowchart for our distance

example in chapter 9 might be



Anything — flowcharts, subroutines, and also **REM** statements — that makes the program clearer gives you a better understanding of it; and then you are sure to write fewer bugs. But subroutines also help you get out the bugs you've already written, by making the program easier to test. You will find it much easier to test the subroutines individually and make sure they fit together properly than to test a whole unstructured program.

Subroutines, then, help with the box "find the bugs," and this is the box where you need all the help you can get, for it is often the most exasperating. Other hints for finding bugs are

(i) Make sure there are no typing errors. Always do this.

(ii) Try to determine what all the variables should be at each stage — and, if possible, explain them in **REM** statements. You can check the variables at a given point in the program by inserting a **PRINT** statement at that point.

(iii) If the effect of the program is to make the program stop with an error report, use the information as thoroughly as you can. Look up the report code and decide why it stopped on a particular line. Print out the values of the variables, if necessary.

(iv) You might be able to step through a program line by line by typing in its lines as commands.

(v) Pretend to be the computer: run the program on yourself, using pencil and paper to note down the values of the variables.

Once you've found the bugs, fixing them is much like writing the original program, but you must test the program again. It is surprisingly easy to fix one bug only to introduce another.

Exercises

1. Occasionally — this may have already happened to you — you will lose a lengthy program you have written, or almost finished, by disconnecting the power supply accidentally. This sort of thing sometimes happens spontaneously; it is not a bug but a *glitch*. There is nothing you can do about it. If it happens much too often, there is probably something wrong, but it would be worth saving the incomplete program on tape halfway through.

2. The flowchart for the distance calculator has no "finish" box. Does this matter? Where would you put one in the flowchart if you wanted to?

Chapter 23

Number Systems

We believe that humans count by tens — the *decimal* system — because humans have ten fingers. Computers count internally "by twos", or in *binary*. This is not because they have only two fingers, but because they can only distinguish between two states of their many internal switches; on or off, 1 or Ø.

Although engineers use a binary system when building computers (see right column of table next page), another number system is useful because it is easier to read and can be easily converted to binary. It is called *hexadecimal* (by sixteens, or having *base sixteen*), and begins:

Hex	*English*
0	zero
1	one
2	two
:	:
:	:
9	nine
A	ten
B	eleven
C	twelve
D	thirteen
E	fourteen
F	fifteen
1Ø	sixteen
:	:
:	:
19	twenty-five
1A	twenty-six
1B	twenty-seven
:	:
:	:
1F	thirty-one
2Ø	thirty-two
21	thirty-three
:	:
:	:
9E	one hundred and fifty-eight
9F	one hundred and fifty-nine
AØ	one hundred and sixty
A1	one hundred and sixty-one
:	:
:	:
FE	two hundred and fifty-four
FF	two hundred and fifty-five
1ØØ	two hundred and fifty-six

If you are using hex notation and you want to make the fact quite clear, write "h" at the end of the number and say "hex". For instance, for one hundred and fifty-eight, write "9Eh" and say "nine E hex".

In the different systems, counting begins

English	*Decimal*	*Hexadecimal*	*Binary*
zero	Ø	Ø	Ø or ØØØØ
one	1	1	1 or ØØØ1
two	2	2	1Ø or ØØ1Ø
three	3	3	11 or ØØ11
four	4	4	1ØØ or Ø1ØØ
five	5	5	1Ø1 or Ø1Ø1
six	6	6	11Ø or Ø11Ø
seven	7	7	111 or Ø111
eight	8	8	1ØØØ
nine	9	9	1ØØ1
ten	1Ø	A	1Ø1Ø
eleven	11	B	1Ø11
twelve	12	C	11ØØ
thirteen	13	D	11Ø1
fourteen	14	E	111Ø
fifteen	15	F	1111
sixteen	16	1Ø	1ØØØØ

The important point is that sixteen is equal to two raised to the fourth power, which makes converting between hex and binary very easy.

To convert hex to binary, change each hex digit into four bits, using the table above. The binary digits Ø and 1 are referred to as *bits*.

To convert binary to hex, divide the binary number into groups of four bits, starting on the right, and then change each group into the corresponding hex digit.

The bits inside the computer are mostly grouped into sets of eight, or *bytes*. A single byte can represent any number from zero to two hundred and fifty-five (11111111 binary or FF hex); or, alternatively, any character in the T/S 1ØØØ character set. Its value can be written with two hex digits.

Two bytes can be grouped together to make what is technically called a *word*. A word can be written using sixteen bits or four hex digits, and represents a number from Ø to (in decimal) $2^{16}-1 = 65535$.

A byte is always eight bits, but words vary from computer to computer.

Summary

Decimal, hexadecimal and binary systems.
Bits and bytes (don't confuse them) and words.

Exercises

1. How would you convert from pounds to ounces and back again
   (i) when all the numbers are written in decimal?
   (ii) when all the numbers are written in hex?

2. How would you convert between decimal and hex? (Hint: exercise 1.)
Write programs on the T/S 1000 to convert numerical values into the strings giving their hex representation, and vice versa. (This is what **STR$** and **VAL** do with decimal representations.)

Chapter 24

How the Computer Works

It is beyond the scope of this manual to describe in detail the electronics of the T/S 1ØØØ and its operation, but we can give some idea of the purpose of its larger components.

The illustration in this chapter shows the T/S 1ØØØ with its outer case removed. The rectangular black components with the metal legs are ICs — integrated circuits. Actually you are looking at only the package — the IC itself is much smaller.

The most important IC of the T/S 1ØØØ is the CPU (Central Processing Unit). It is a Z8ØA microprocessor. The processor does the arithmetic, and electronically controls the rest of the computer according to the operating system program.

The operating system is contained in the ROM IC. This is a solid state electronic storage device which has a program permanently wired in to make the CPU work. The program is unique to the T/S 1ØØØ. In symbolic form it is a long sequence of bytes. Each byte has an *address* showing its position in

the ROM — the first one has address Ø, the second has address 1, and so on up to 8191; which is why the Sinclair BASIC is called 8K BASIC.

You can see what byte is at a given address by using the function **PEEK**. For example, this program prints out the first 21 bytes in the ROM (and their addresses.)

```
1Ø PRINT "ADDRESS";TAB 8;"BYTE"
2Ø FOR A=Ø TO 2Ø
3Ø PRINT A;TAB 8;PEEK A
4Ø NEXT A
```

The RAM chip is an electronic "scratch pad" which is hooked up to the CPU. The BASIC programs that you type in are stored electronically here, as are its variables, the television picture, and the system variables (another technical word!).

Like the ROM, the RAM storage is arranged into bytes, each with an address. These range from 16384 to 18431 (or 32767 if you have a 16K RAM pack extension). As with the ROM you can find the values of these bytes by using **PEEK**. The difference is that you can also change them.

Type

```
POKE 17ØØØ,57
```

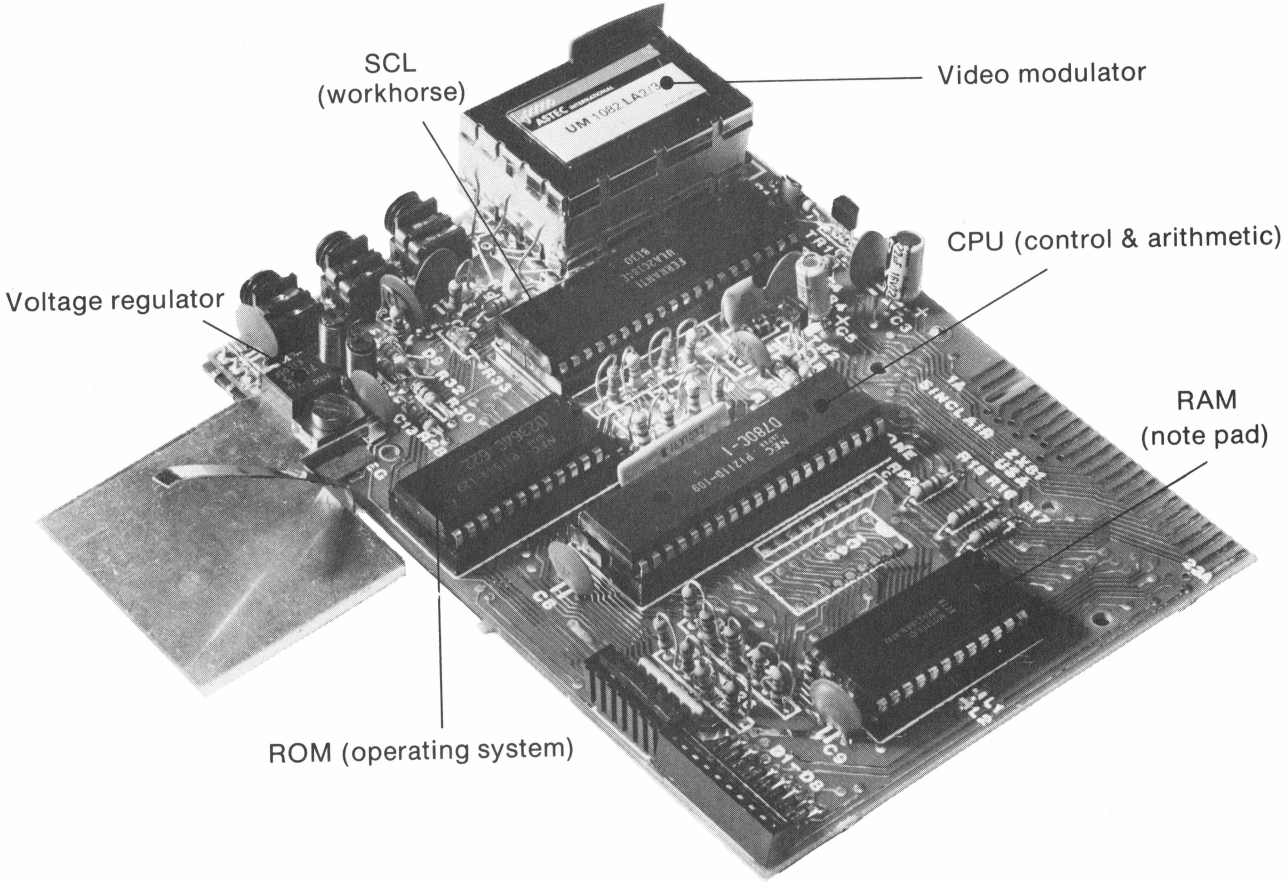
This gives the byte at address 17ØØØ the value 57. If you now type

```
PRINT PEEK 17ØØØ
```

you get your number 57 back. (Try poking in other values.)

Note that the address has to be between Ø and 65535; and most of these will refer to bytes in ROM or nowhere at all, and so have no effect. The value must be between −255 and +255, and if it is negative it gets 256 added to it.

The ability to poke gives you immense power over the computer if you know how to use it; however, the necessary knowledge is rather more than can be imparted in an introductory manual like this.


SCL
(workhorse)
Video modulator
CPU (control & arithmetic)
Voltage regulator
RAM
(note pad)
ROM (operating system)

The last large IC we call the SCL (Sinclair Computer Logic). It is a ULA (more technical jargon) specially designed and made for the T/S 1000. It wires the other components to one another in an ingenious way.

The modulator converts the computer's television output into a form suitable for the television, and the regulator converts the smoothed, but unregulated 9 volts of the power supply to a regulated 5 volts.

Summary

Chips
Statements: **POKE**
Functions: **PEEK**

Chapter 25

Using Machine Code

This chapter is written for those who understand Z80 machine code, the set of instructions that the Z80 processor chip uses. If you wish to learn more about Z80 machine codes and programming, the following books published by Reston Publishing Company, Inc., Reston, Virginia, may be helpful: *Mastering Machine Code on Your ZX81*, by Toni Baker; and *Z-80 Users Manual*, by Joseph J. Carr.

The ultimate authority is the *Z80 Assembly Language Programming Manual*, together with the *Z80-CPU, Z80A-CPU Technical Manual*, published by Zilog; but these can hardly be recommended for beginners.

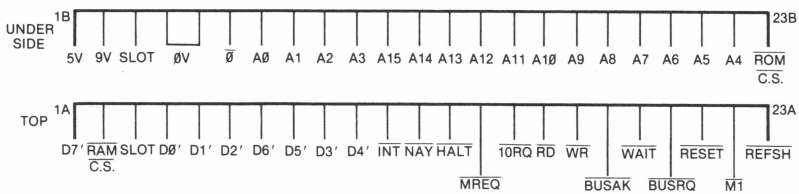
Machine code routines can be executed from within a BASIC program by using the function **USR**. The argument of **USR** is the starting address of the routine, and its result is a two-byte unsigned integer, the contents of the bc register pair on return. The return address to the BASIC is stacked in the usual way, so return is by a Z80 **RET** instruction.

There are certain restrictions on **USR** routines:

(i) On return, the i register must have the value 1Eh.

(ii) The display routine uses a',f',ix,iy and r registers, a **USR** routine should not use these if compute and display is operating. (It is not even safe to read the af' pair.)

All these lines from the processor are exposed at the back of the T/S 1ØØØ, so in principle you can do anything with a T/S 1ØØØ that you can with a Z8Ø. The T/S 1ØØØ hardware might sometimes get in the way, though, especially in compute and display. Here is a diagram of the exposed connections at the back:



A piece of machine code in the middle of memory runs the risk of being overwritten by the BASIC system. Some safer places are

(i) In a **REM** statement: type in a **REM** statement with enough characters to hold your machine code, which you then poke in. Avoid halt instructions, since these will be recognized as the end of the **REM** statement.

(ii) In a string: set up a long enough string, and then assign a machine code byte to each character.

In both of these the code is safe, but likely to move about; this is especially the case with strings. In the Appendix, the character set, you will find the characters and Z8Ø instructions written side by side in order, and you may well find this useful when entering code.

(iii) At the top of the memory. When theT/S 1ØØØ is switched on, it tests to see how much memory there is and puts the machine stack right at the top so that there is no space for **USR** routines there. It stores the address of the first nonexistent byte (e.g., 17K, or 174Ø8, if you have 1K memory) in a system variable known as RAMTOP, in the two bytes with addresses 16388 and 16389. **NEW**, on the other hand, does not do a full memory test, but only checks up as far as just before the address in RAMTOP. Thus, if you poke the address of an existing byte into RAMTOP, for **NEW** all the memory from that byte on is outside the BASIC system and is left alone. For instance, suppose you have 1K memory and you have just switched on the computer.

PRINT PEEK 16388+256***PEEK** 16389

tells you the address (174Ø8) of the first nonexistent byte.

Now suppose you have a **USR** routine 2Ø bytes long. You want to change RAMTOP to 17388 = 236 + 256*67 (how would you work this out in the computer?), so type

POKE 16388,236
POKE 16389,67

and then **NEW**. The twenty bytes of memory from address 17388 to 17407 are now yours to do with what you like. If you then type **NEW** again it will not affect these twenty bytes.

The top of memory is a good place for **USR** routines, safe (even from **NEW**) and immobile. Its main disadvantage is that it is not saved by **SAVE**.

Summary

Functions: **USR**
Statements: **NEW**

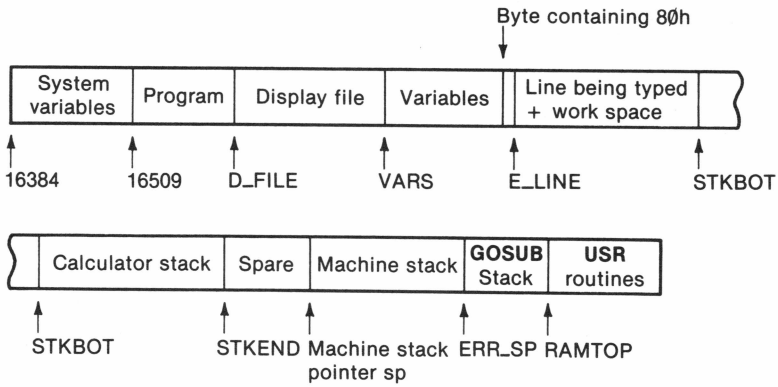
Exercises

1. Make RAMTOP equal to 16700 and then execute **NEW**. You will get an idea of what happens when the memory gets full.

Chapter 26

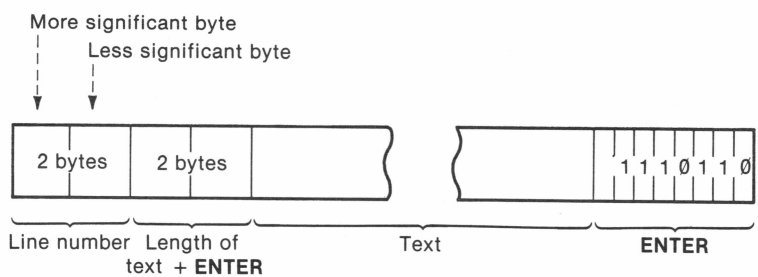
Organization of Storage

The memory is divided into different areas for storing different kinds of information. The areas are only large enough for the information they actually contain, and if you insert more at a given point (for instance, by adding a program line or variable), space is made by shifting up everything above that point. Conversely, if you delete information, everything above the deletion is shifted down.



The system variables contain various pieces of information that tell the computer what sort of state the computer is in. They are listed fully in the next chapter, but for the moment note that there are some (called D_FILE, VARS, E_LINE and so on) that contain the addresses of the boundaries between the various areas in the memory. These are not BASIC variables, and their names will not be recognized by the computer.

In the program, each line is stored as



Note that, in contrast with all other cases of two-byte numbers in the Z8Ø, the line number here (and also in a **FOR-NEXT** control variable) is stored with its more significant byte first: that is to say, in the order that you would write them down.

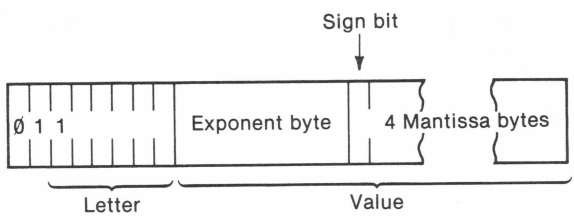
A numerical constant in the program is followed by its binary form, using the character **CHR$** 126 followed by five bytes for the number itself.

The display file is the memory copy of the television picture. It begins with an **ENTER** character, followed by twenty-four lines of text, each finishing with an **ENTER**. The system is so designed that a line of text does not need a full thirty-two characters: final spaces can be omitted. This is done to save space when the memory is small.

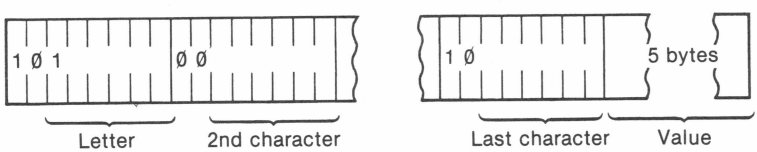
When the total amount of memory (according to the system variable RAMTOP) is less than 3 1/4K, a clear screen — as set up at the start or by **CLS** — consists of just 25 **ENTER**s. When the memory is bigger, a clear screen is padded out with 24*32 spaces, and on the whole it stays at its full size; **SCROLL**, however, and certain conditions where the lower part of the screen expands to more than two lines, can upset this by introducing short lines at the bottom.

The variables have different formats, depending on their different natures.

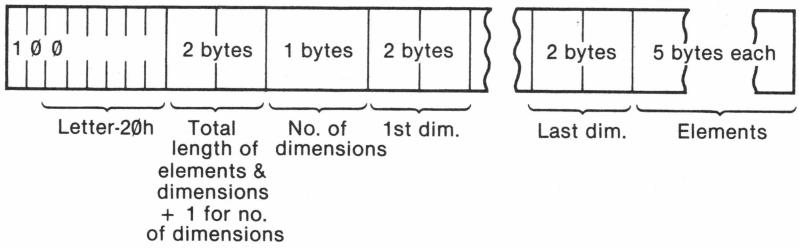
Number whose name is one letter only:



Number whose name is longer than one letter:



Array of numbers:

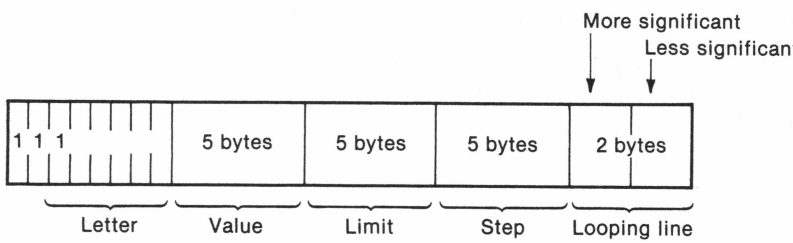


The order of the elements is:
first, the elements for which the first subscript is 1
next, the elements for which the first subscript is 2
next, the elements for which the first subscript is 3
and so on for all possible values of the first subscript.

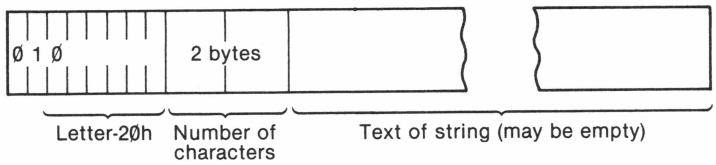
The elements with a given first subscript are ordered in the same way using the second subscript, and so on down to the last.

As an example, the elements of the 3 X 6 array B in Chapter 22 are stored in the order B(1,1), B(1,2), B(1,3), B(1,4), B(1,5), B(1,6), B(2,1), B(2,2),. . ., B(2,6), B(3,1), B(3,2),. . ., B(3,6).

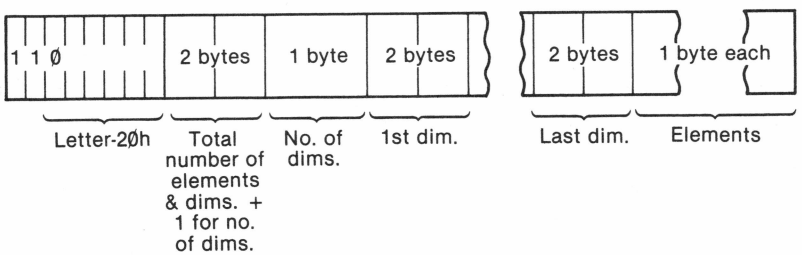
Control variable of a **FOR-NEXT** loop:



String:



Array of characters:



The part starting at E_LINE contains the line being typed (as a command, a program line, or **INPUT** data) and also some work space.

The calculator is the part of the BASIC system that deals with arithmetic, and the numbers it is operating are held mostly in the calculator stack.

The spare part contains the space so far unused.

The machine stack is the stack used by the Z-8Ø chip to hold return addresses and so on.

The space for **USR** routines has to be set aside by you, using **NEW** as described in the last chapter.

Chapter 27

The System Variables

The bytes in memory from 16384 to 16508 are set aside for specific uses by the system. You can peek them to find out various things about the system, and some of them can be usefully poked. They are listed here with their uses.

These are called system variables and carry names, but do not confuse them with the variables used by the BASIC. You cannot use the names in a BASIC program; they are simply mnemonics that are used to make it easier to refer to the variables.

The abbreviations in column 1 have the following meanings.

X The variable should not be poked, because the system might crash.
N Poking the variable will have no lasting affect.
S The variable is saved by **SAVE**.

The number in column 1 is the number of bytes in the variable. For two bytes, the first one is the *less* significant byte — the reverse of what you might expect. So to poke a value v to a two-byte variable at address n, use

POKE n,v−256***INT** (v/256)
POKE n+1,**INT** (v/256)

and to peek its value, use the expression:

PEEK n + 256***PEEK** (n+1)

Notes	*Address*	*Name*	*Contents*
1	16384	ERR_NR	1 less than the report code. Starts off at 255 (for −1), so **PEEK** 16384, if it works at all, gives 255. **POKE** 16384, n can be used to force an error halt: Ø <= n <= 14 gives one of the usual reports, 15 <= n <= 34 or 99 <= n <= 127 gives a nonstandard report, and 35 <= n <= 98 is likely to mess up the display file.
X1	16385	FLAGS	Various flags to control the BASIC system.
X2	16386	ERR_SP	Address of first item on machine stack (after **GOSUB** returns).
2	16388	RAMTOP	Address of first byte above BASIC system area. You can poke this to make **NEW** reserve space above that area (see Chapter 25) or to fool **CLS** into setting up a minimal display file (Chapter 26).
N1	16390	MODE	Specifies K, L, F or G cursor
N2	16391	PPC	Line number of statement currently being executed. Poking this has no lasting effect except in the last line of the program.
S1	16393	VERSN	Ø identifies 8K ROM in saved programs.
S2	16394	E_PPC	Number of current line (with program cursor).
SX2	16396	D_FILE	See Chapter 26.
S2	16398	DF_CC	Address of **PRINT** position in display file. Can be poked so that **PRINT** output is sent elsewhere.
SX2	16400	VARS	See Chapter 26.
SN2	16402	DEST	Address of variable in assignment.
SX2	16404	E_LINE	See Chapter 26.
SX2	16406	CH_ADD	Address of the next character to be interpreted: the character after the argument of **PEEK**, or the **ENTER** at the end of a **POKE** statement.
S2	16408	X_PTR	Address of the character preceding the marker.

Notes	*Address*	*Name*	*Contents*
SX2	16410	STKBOT	See Chapter 26.
SX2	16412	STKEND	
SN1	16414	BREG	Calculator's b register.
SN2	16415	MEM	Address of area used for calculator's memory. (Usually MEMBOT, but not always.)
S1	16417	not used	
SX1	16418	DF_SZ	The number of lines (including one blank line) in the lower part of the screen.
S2	16419	S_TOP	The number of the top program line in automatic listings.
SN2	16421	LAST_K	Shows which keys pressed
SN1	16423		Debounce status of keyboard.
SN1	16424	MARGIN	Number of blank lines above or below picture — 31.
SX2	16425	NXTLIN	Address of next program line to be executed.
S2	16427	OLDPPC	Line number to which **CONT** jumps.
SN1	16429	FLAGX	Various flags.
SN2	16430	STRLEN	Length of string type designation in assignment.
SN2	16432	T-ADDR	Address of next item in syntax table (very unlikely to be useful).
S2	16434	SEED	The seed for **RND**. This is the variable that is set by **RAND**.
S2	16436	FRAMES	Counts the frames displayed on the television. Bit 15 is 1. Bits 0 to 14 are decremented for each frame sent to the television. This can be used for timing, but **PAUSE** also uses it. **PAUSE** resets bit 15 to 0 and puts in bits 0 to 14 the length of the pause. When these have been counted down to zero, the pause stops. If the pause stops because of a key depression, bit 15 is set to one again.
S1	16438	COORDS	x-coordinate of last point **PLOT**ted.
S1	16439		y-coordinate of the last point **PLOT**ted.
S1	16440	PR_CC	Less significant byte of address of next position for **LPRINT** to print at (in PRBUFF).
SX1	16441	S_POSN	Column number for **PRINT** position.
SX1	16442		Line number for **PRINT** position.

Notes	*Address*	*Name*	*Contents*
S1	16443	CDFLAG	Various flags. Bit 7 is on (1) during compute and display e.
S33	16444	PRBUFF	Printer buffer (33rd character) is **ENTER**.
SN3Ø	16477	MEMBOT	Calculator's memory area; used to store numbers that cannot conveniently be put on the calculator stack.
S2	165Ø7	not used	

Exercises

1. Try this program

```
1Ø FOR N=Ø TO 21
2Ø PRINT PEEK (PEEK 164ØØ+256* PEEK 164Ø1 + N)
3Ø NEXT N
```

This tells you the first 22 bytes of the variables area: try to match up the control variable N with the description in Chapter 26.

2. In the program above, change line 2Ø to

```
2Ø PRINT PEEK (165Ø9+N)
```

This tells you the first 22 bytes of the program area. Match these up with the program itself.

Appendix

The Character Set

This is the complete T/S 1ØØØ character set, with codes in decimal and hex. If one imagines the codes being Z8Ø machine code instructions, then the right-hand columns give the corresponding assembly language mnemonics. As you are probably aware, certain Z8Ø instructions are compounds starting with CBh or EDh, as shown in the right-hand columns.

Code	*Character*	*Hex*	*Z8Ø assembler*	*-after CB*	*-after ED*
Ø	space	ØØ	nop	ric b	
1	▘	Ø1	ld bc,NN	ric c	
2	▖	Ø2	ld (bc), a	ric d	
3	▌	Ø3	inc bc	ric e	
4	▗	Ø4	inc b	ric h	

Code	*Character*	*Hex*	*Z80 assembler*	*-after CB*	*-after ED*
5	◧	Ø5	dec b	ric l	
6	▪	Ø6	ld b,N	ric (hl)	
7	◧	Ø7	rlca	ric a	
8	▒	Ø8	ex af,af'	rrc b	
9	▒	Ø9	add hl,bc	rrc c	
1Ø	▒	ØA	ld a,(bc)	rrc d	
11	"	ØB	dec bc	rrc e	
12	£	ØC	inc c	rrc h	
13	$	ØD	dec c	rrc l	
14	:	ØE	ld c,N	rrc (hl)	
15	?	ØF	rrca	rrc a	
16	(	1Ø	djnz DIS	rl b	
17	)	11	ld de,NN	rl c	
18	>	12	ld (de),a	rl d	
19	<	13	inc de	rl e	
2Ø	=	14	inc d	rl h	
21	+	15	dec d	rl l	
22	—	16	id d,N	rl (hl)	
23	*	17	rla	rl a	
24	/	18	jr DIS	rr b	
25	;	19	add hl,de	rr c	
26	'	1A	ld a,(de)	rr d	
27	.	1B	dec de	rr e	
28	Ø	1C	inc e	rr h	
29	1	1D	dec e	rr l	
3Ø	2	1E	ld e,N	rr (hl)	
31	3	1F	rra	rr a	
32	4	2Ø	jr nz,DIS	sla b	
33	5	21	ld hl,NN	sla c	
34	6	22	id (NN),hl	sla d	
35	7	23	inc hl	sla e	
36	8	24	inc h	sla h	
37	9	25	dec h	sla l	
38	A	26	ld h,N	sla (hl)	
39	B	27	daa	sla a	
4Ø	C	28	jr z,DIS	sra b	
41	D	29	add hl,hl	sra c	
42	E	2A	ld hl,(NN)	sra d	
43	F	2B	dec hl	sra e	
44	G	2C	inc l	sra h	
45	H	2D	dec l	sra l	
46	I	2E	ld l,N	sra (hl)	
47	J	2F	cpl	sra a	
48	K	3Ø	jr nc,DIS		
49	L	31	ld sp,NN		
5Ø	M	32	id (NN),a		

Code	*Character*	*Hex*	*Z80 assembler*	*-after CB*	*-after ED*
51	N	33	inc sp		
52	O	34	inc (hl)		
53	P	35	dec (hl)		
54	Q	36	id (hl),N		
55	R	37	scf		
56	S	38	jr c,DIS	srl b	
57	T	39	add hl,sp	srl c	
58	U	3A	ld a,(NN)	srl d	
59	V	3B	dec sp	srl e	
60	W	3C	inc a	srl h	
61	X	3D	dec a	srl l	
62	Y	3E	ld a,N	srl (hl)	
63	Z	3F	ccf	srl a	
64	**RND**	40	ld b,b	bit 0,b	in b,(c)
65	**INKEY$**	41	ld b,c	bit 0,c	out (c),b
66	**PI**	42	ld b,d	bit 0,d	sbc hl,bc
67		43	ld b,e	bit 0,e	ld (NN),bc
68		44	ld b,h	bit 0,h	neg
69		45	ld b,l	bit 0,l	retn
70		46	ld b,(hl)	bit 0,(hl)	im 0
71		47	ld b,a	bit 0,a	ld i,a
72		48	ld c,b	bit 1,b	in c,(c)
73		49	ld c,c	bit 1,c	out (c),c
74		4A	ld c,d	bit 1,d	adc hl,bc
75		4B	ld c,e	bit 1,e	ld bc,(NN)
76		4C	ld c,h	bit 1,h	
77		4D	ld c,l	bit 1,l	reti
78		4E	ld c,(hl)	bit 1,(hl)	
79		4F	ld c,a	bit 1,a	ld r,a
80		50	ld d,b	bit 2,b	in d,(c)
81	not used	51	ld d,c	bit 2,c	out (c),d
82		52	ld d,d	bit 2,d	sbc hl,de
83		53	ld d,e	bit 2,e	ld (NN),de
84		54	ld d,h	bit 2,h	
85		55	ld d,l	bit 2,l	
86		56	ld d,(hl)	bit 2,(hl)	im 1
87		57	ld d,a	bit 2,a	ld a,i
88		58	ld e,b	bit 3,b	in e,(c)
89		59	ld e,c	bit 3,c	out (c),e
90		5A	ld e,d	bit 3,d	adc hl,de
91		5B	ld e,e	bit 3.e	ld de,(NN)
92		5C	ld e,h	bit 3,h	
93		5D	ld e,l	bit 3,l	
94		5E	ld e,(hl)	bit 3,(hl)	im 2
95		5F	ld e,a	bit 3,a	ld a,r
96		60	ld h,b	bit 4,b	in h,(c)

Code	*Character*	*Hex*	*Z80 assembler*	*-after CB*	*-after ED*
97		61	ld h,c	bit 4,c	out (c),h
98		62	ld h,d	bit 4,d	sbc hl,hl
99		63	ld h,e	bit 4,e	ld (NN),hl
100		64	ld h,h	bit 4,h	
101		65	ld h,l	bit 4,l	
102		66	ld h,(hl)	bit 4,(hl)	
103	not used	67	ld h,a	bit 4,a	rrd
104		68	ld l,b	bit 5,b	in l,(c)
105		69	ld l,c	bit 5,c	out (c),l
106		6A	ld l,d	bit 5,d	adc hl,hl
107		6B	ld l,e	bit 5,e	ld de,(NN)
108		6C	ld l,h	bit 5,h	
109		6D	ld l,l	bit 5,l	
110		6E	ld l,(hl)	bit 5,(hl)	
111		6F	ld l,a	bit 5,a	rid
112	cursor up ⌃	70	ld (hl),b	bit 6,b	
113	cursor down ⌄	71	ld (hl),c	bit 6,c	
114	cursor left ◊	72	ld (hl),d	bit 6,d	sbc hl,sp
115	cursor right ◊	73	ld (hl),e	bit 6,e	ld (NN),sp
116	**GRAPHICS**	74	ld (hl),h	bit 6,h	
117	**EDIT**	75	ld (hl),l	bit 6,l	
118	**ENTER**	76	halt	bit 6, (hl)	
119	**DELETE**	77	ld (hl),a	bit 6,a	
120	K / L mode	78	ld a,b	bit 7,b	in a,(c)
121	**FUNCTION**	79	ld a,c	bit 7,c	out (c),a
122	not used	7A	ld a,d	bit 7,d	adc hl,sp
123	not used	7B	ld a,e	bit 7,e	ld sp,(NN)
124	not used	7C	ld a,h	bit 7,h	
125	not used	7D	ld a,l	bit 7,l	
126	number	7E	ld a,(hl)	bit 7,(hl)	
127	cursor	7F	ld a,a	bit 7,a	
128	■	80	add a,b	res Ø,b	
129	▟	81	add a,c	res Ø,c	
130	▐	82	add a,d	res Ø,d	
131	▄	83	add a,e	res Ø,e	
132	▙	84	add a,h	res Ø,h	
133	▌	85	add a,l	resØ,l	
134	▚	86	add a,(hl)	res Ø,(hl)	
135	▗	87	add a,a	res Ø,a	
136	▒	88	adc a,b	res 1,b	
137	▜	89	adc a,c	res 1,c	
138	▀	8A	adc a,d	res 1,d	
139	inverse "	8B	adc a,e	res 1,e	
140	inverse	8C	adc a,h	res 1,h	
141	inverse $	8D	adc a,l	res 1,l	
142	inverse :	8E	adc a,(hl)	res 1,(hl)	

Code	*Character*	*Hex*	*Z80 assembler*	*-after CB*	*-after ED*
143	inverse ?	8F	adc a,a	res 1,a	
144	inverse (	90	sub b	res 2,b	
145	inverse)	91	sub c	res 2,c	
146	inverse >	92	sub d	res 2,d	
147	inverse <	93	sub e	res 2,e	
148	inverse =	94	sub h	res 2,h	
149	inverse +	95	sub l	res 2,l	
150	inverse –	96	sub (hl)	res 2,(hl)	
151	inverse *	97	sub a	res 2,a	
152	inverse /	98	sbc a,b	res 3,b	
153	inverse ;	99	sbc a,c	res 3,c	
154	inverse ,	9A	sbc a,d	res 3,d	
155	inverse .	9B	sbc a,e	res 3,e	
156	inverse 0	9C	sbc a,h	res 3,h	
157	inverse 1	9D	sbc a,l	res 3,l	
158	inverse 2	9E	sbc a,(hl)	res 3,(hl)	
159	inverse 3	9F	sbc a,a	res 3,a	
160	inverse 4	A0	and b	res 4,b	ldi
161	inverse 5	A1	and c	res 4,c	cpi
162	inverse 6	A2	and d	res 4,d	ini
163	inverse 7	A3	and e	res 4,e	outi
164	inverse 8	A4	and h	res 4,h	
165	inverse 9	A5	and l	res 4,l	
166	inverse A	A6	and (hl)	res 4,(hl)	
167	inverse B	A7	and a	res 4,a	
168	inverse C	A8	xor b	res 5,b	ldd
169	inverse D	A9	xor c	res 5,c	cpd
170	inverse E	AA	xor d	res 5,d	ind
171	inverse F	AB	xor e	res 5,e	outd
172	inverse G	AC	xor h	res 5,h	
173	inverse H	AD	xor l	res 5,l	
174	inverse I	AE	xor (hl)	res 5,(hl)	
175	inverse J	AF	xor a	res 5,a	
176	inverse K	B0	or b	res 6,b	ldir
177	inverse L	B1	or c	res 6,c	cpir
178	inverse M	B2	or d	res 6,d	inir
179	inverse N	B3	or e	res 6,e	otir
180	inverse O	B4	or h	res 6,h	
181	inverse P	B5	or l	res 6,l	
182	inverse Q	B6	or (hl)	res 6,(hl)	
183	inverse R	B7	or a	res 6,a	
184	inverse S	B8	cp b	res 7,b	lddr
185	inverse T	B9	cp c	res 7,c	cpdr
186	inverse U	BA	cp d	res 7,d	indr
187	inverse V	BB	cp e	res 7,e	otdr
188	inverse W	BC	cp h	res 7,h	

Code	*Character*	*Hex*	*Z80 assembler*	*-after CB*	*-after ED*
189	inverse X	BD	cp l	res 7,l	
190	inverse Y	BE	cp (hl)	res 7,(hl)	
191	inverse Z	BF	cp a	res 7,a	
192	"""	C0	ret nz	set 0,b	
193	**AT**	C1	pop bc	set 0,c	
194	**TAB**	C2	jp nz,NN	set 0,d	
195	not used	C3	jpNN	set 0,e	
196	**CODE**	C4	call nz,NN	set 0,h	
197	**VAL**	C5	push bc	set 0,l	
198	**LEN**	C6	add a,N	set 0,(hl)	
199	**SIN**	C7	rst 0	set 0,a	
200	**COS**	C8	ret z	set 1,b	
201	**TAN**	C9	ret	set 1,c	
202	**ASN**	CA	jp z,NN	set 1,d	
203	**ACS**	CB		set 1,e	
204	**ATN**	CC	call z,NN	set 1,h	
205	**LN**	CD	call NN	set 1,l	
206	**EXP**	CE	adc a,N	set 1,(hl)	
207	**INT**	CF	rst 8	set 1,a	
208	**SQR**	D0	ret nc	set 2,b	
209	**SGN**	D1	pop de	set 2,c	
210	**ABS**	D2	jp nc,NN	set 2,d	
211	**PEEK**	D3	out N,a	set 2,e	
212	**USR**	D4	call nc,NN	set 2,h	
213	**STR$**	D5	push de	set 2,l	
214	**CHR$**	D6	sub N	set 2,(hl)	
215	**NOT**	D7	rst 16	set 2,a	
216	**	D8	ret c	set 3,b	
217	**OR**	D9	exx	set 3,c	
218	**AND**	DA	jp c,NN	set 3,d	
219	<=	DB	in a,N	set 3,e	
220	>=	DC	call c,NN	set 3,h	
221	<>	DD	prefixes instructions using ix	set 3,l	
222	**THEN**	DE	sbc a,N	set 3,(hl)	
223	**TO**	DF	rst 24	set 3,a	
224	**STEP**	E0	ret po	set 4,b	
225	**LPRINT**	E1	pop hl	set 4,c	
226	**LLIST**	E2	jp po,NN	set 4,d	
227	**STOP**	E3	ex (sp),hl	set 4,e	
228	**SLOW**	E4	call po,NN	set 4,h	
229	**FAST**	E5	push hl	set 4,l	
230	**NEW**	E6	and N	set 4,(hl)	
231	**SCROLL**	E7	rst 32	set 4,a	
232	**CONT**	E8	ret pe	set 5,b	
233	**DIM**	E9	jp (hl)	set 5,c	

Code	*Character*	*Hex*	*Z80 assembler*	*-after CB*	*-after ED*
234	**REM**	EA	jp pe,NN	set 5,d	
235	**FOR**	EB	ex de,hl	set 5,e	
236	**GOTO**	EC	call pe,NN	set 5,h	
237	**GOSUB**	ED		set 5,l	
238	**INPUT**	EE	xor N	set 5,(hl)	
239	**LOAD**	EF	rst 40	set 5,a	
240	**LIST**	F0	ret p	set 6,b	
241	**LET**	F1	pop af	set 6,c	
241	**PAUSE**	F2	jp p,NN	set 6,d	
243	**NEXT**	F3	di	set 6,e	
244	**POKE**	F4	call p,NN	set 6,h	
245	**PRINT**	F5	push af	set 6,l	
246	**PLOT**	F6	or N	set 6,(hl)	
247	**RUN**	F7	rst 48	set 6,a	
248	**SAVE**	F8	ret m	set 7,b	
249	**RAND**	F9	ld sp,hl	set 7,c	
250	**IF**	FA	jp m,NN	set 7,d	
251	**CLS**	FB	ei	set 7,e	
252	**UNPLOT**	FC	call m,NN	set 7,h	
253	**CLEAR**	FD	prefixes instructions using iy	set 7,l	
254	**RETURN**	FE	cp N	set 7,(hl)	
255	**COPY**	FF	rst 56	set 7,a	

Index

This index includes the keys on the keyboard and how to obtain them (the mode — K, L, F or G — and whether shifted or not), and their codes.

Usually an entry is referenced only once per chapter, so having found one reference, look through the rest of the chapter including the exercises.

A

ABS	F, on G. Code 210	70
accuracy		54
ACS	F, on S. Code 203. Arccosine	70
addition of strings		86
address		
—of a byte		120
return address		41
alphabetical order		33,61
AND	K or L, shifted 2. Code 218.	33
antilog		54
argument		70
arithmetic expression		53
array		79
ASN	F, on A. Code 202. Arcsine.	70
assign		27
AT	F, on C. Code 193.	55,87,94
ATN	F, on D. Code 204. Arctangent.	70

B

bar chart		60
BASIC		21,97
binary		
—operation		35,52
—system		115
bit		117
BOLD TYPE		16,97
BREAK	On **SPACE**. Only recognized as **BREAK** in certain situations.	11,22
buffer		136
bug		111

byte 117

C

call 41
cassette recorder 6,9
character 57
—positon 63
—set 57,137
control character 60
CHR$ F, on U. Code 214. 57
CLEAR K, on X. Code 253. 29
CLS K, on V. Code 251. 56
CODE F, on I. Code 196. 57
code 57
machine code 123
command 15
comparison 32
—of numbers 32
—of strings 33
compute & display 40
concatenation 86
condition 32
conditional expression 36
CONT K, on C. Code 232. 22
control
—character 60
—variable 27
coordinate 63
COPY K, on Z. Code 255. 93
COS F, on W. Code 200. 70
CPU 119
cursor 8
F cursor 69
G cursor 58
K cursor 8
L cursor 10
program cursor (>) 24

D

DATA 82
decimal system 115
degree 72
DELETE G, K or L, shifted 0. Code 119 19,58
DIM K, on D. Code 233. 47,79

dimension		47,79
display file		47,129
E		
—in exponent part		53
EDIT	K or L, shifted 1. Code 117.	24
element		79
empty string		57,88
ENTER	Code 118.	13,17,21
entry point		45
execute		22
EXP	F, on X. Code 206.	70
exponent		53
—byte		129
—part		53
expression		53
arithmetic expression		53
conditional expression		36
logical expression		33
numeric expression		53
string expression		87
F		
F mode		24,69
false		32
FAST	K or L, shifted F. Code 229.	39
fast mode		39
floating point		52
flowchart		112
FOR	K, on F. Code 235.	28
FUNCTION	K or L, shifted **ENTER**. Code 121.	24,69
function		
—mode		69
G		
G mode		58
glitch		114
GOSUB	K, on H. Code 237.	41
—stack		41
GOTO	K, on G. Code 236.	22,41
graph		64
graphics		58,63
—symbol		58

—mode 58
GRAPHICS G, K or L shifted 9. Code 116. 58
gray characters 60

H

hex 116
hexadecimal 116

I

if 28,31
IF K on U. Code 250. 28,31
INKEY$ F, on B. Code 65. 76
INPUT K, on I. Code 238. 30
INT F, on R. Code 207. 70,72
integer 54,72
interactive 21
inverse
—functions 71
—video 16,58
item 86
PRINT item 86

J

Jack plug 5

K

K mode 16,98
keyboard 15
keyword 16,98
—mode 16,98

L

L mode 16,98
LEFT$ 91
LEN K, on K. Code 198. 86
LET K, on L. Code 241. 27
lettermode 16,98
line
—number 64
program line 18
LIST K, on K. Code 240. 12,28

		28
LLIST	K or L, shifted G. Code 226.	93
LN	F, on Z. Code 205.	70
LOAD	K, on J. Code 239.	10
logic		
—chip		122
—al operation		33
logs		54,71
loop		27
LPRINT	K or L, shifted S. Code 225.	93

M

machine code		123
main program		42,43
mantissa		54,129
memory		47
—expansion board		49
MID$		91
mode		
compute & display		40
fast mode		40
function mode (F)		69
graphics mode (G)		58
keyword mode (K)		16,98
letter mode (L)		16,98
modulo		55,72
modulus		70

N

name		
—of a variable		27
—of a program		10,12
nesting		29
NEW	K, on A. Code 230.	11,21,124
NEXT	K, on N. Code 243.	28
NOT	F, on N. Code 215.	33
nullary operation		70
null string		88
numeric		
—expression		53

O

ON		37

operand 51
operation 51
  binary operation 52
  logical operation 33
  nullary operation 70
  unary operation 52
OR K or L, shifted W. Code 217. 33

P

parentheses 52
PAUSE K, on M. Code 242. 75
PEEK F, on O. Code 211. 120
PI F, on M. Code 66. 30,70
pixel 63
PLOT K, on Q. Code 246. 63
POKE K, on O. Code 244. 120
position 63
  LPRINT position 94
  PRINT position 55
power 51
  —supply 6
PRINT K, on P. Code 245. 15,55
  —item 86
  —position 55
printer 93
priority 52
processor 119
Procrustean assignment 90
program 21
  —cursor 24
  —line 18
pseudorandom 70

Q

quote 10,23,85
  —image 85
  string quote 85

R

radian 72
RAM 120
RAMTOP 124
RAND K, on T. Code 249. 70

random		70
READ		82
recursive		44
register		123
relation		33
REM	K, on E. Code 234.	13,30,113
report		11,18,113,153
RESTORE		82
RETURN	K, on Y. Code 254.	41
return address		41
RIGHT$		91
RND	F, on T. Code 64.	70
ROM		119
rounding	(inc. to nearest integer)	54,61,72
RUN	K, on R. Code 247.	11,22

S

SAVE	K, on S. Code 248.	13
scientific notation		53
SCROLL	K, on B. Code 231.	56
SGN	F, on F. Code 209.	70
shift		10,16
—ed key		16
simple variable		79
SIN	F, on Q. Code 199.	70
slice		89
SLOW	K or L, shifted D. Code 228.	39,40
SQR	F, on H. Code 208.	69
stack		124
calculator stack		128
GOSUB stack		41,128
machine stack		128
statement		21
STEP	K or L, shifted E. Code 224.	29
STOP	K or L, shifted A. Code 227.	28,33
STR$	F, on Y. Code 213.	87
string		23,85
—addition		86
—expression		87
—quote		85
—variable		27,86
subroutine		41
subscript		79
—ed variable		79
—error		90

substring 89
symbol 57
—graphics symbol 58
syntax 19
system variable 133

T
TAB F, on P. Code 194. 24,55,94
TAN F, on E. Code 201. 70
tape
—recorder 6,9
—storage 9
television 5
THEN K or L, shifted 3. Code 222. 28,32
TL$ 91
TO K or L, shifted 4. Code 223. 28,89
token 57
trigonometric functions 70
true 32

U
unary operations 52
unless 36
UNPLOT K, on W. Code 252. 63
USR F, on L. Code 212. 123

V
VAL F, on J. Code 197. 86
value 27
variable 27,86
control variable 27
simple variable 79
string variable 27,86
subscripted variable 79
system variable 133

W
word 117

X
x-coordinate 63

Y
y-coordinate 63

Z
Z-8ØA 119

. K or L, Code 27. Full stop or decimal point. 53
, K or L, shifted full stop. Code 26. Comma. 15,53
; K or L, shifted X. Code 25. Semicolon 16,53
: K or L, shifted Z. Code 14. Colon.
? K or L, shifted C. Code 15. Question mark.
" K or L, shifted P. Code 11. String quote. 85
"" K or L, shifted Q. Code 192. Quote image. 85
(K or L, shifted I. Code 16. Open parenthesis. 52
) K or L, shifted O. Code 17. Close parenthesis. 52
£ K or L, shifted space. Code 12. Pound. 58
$ K or L, shifted U. Code 13. Dollar/String. 86
+ K or L, shifted K. Code 21. Plus. 16,51
- K or L, shifted J. Code 22. Minus. 51
* K or L, shifted B. Code 23. Times. 51
/ K or L, shifted V. Code 24. Divide. 51
** K or L, shifted H. Code 216. To power. 51
= K or L, shifted L. Code 2Ø. Equals. 27,31
> K or L, shifted M. Code 18. Greater than. 32
< K or L, shifted N. Code 19. Less than. 32
<= K or L, shifted R. Code 219. Less than or equal to. 32
>= K or L, shifted Y. Code 22Ø. Greater than or equal to. 32
<> K or L, shifted T. Code 221. Not equal to. 32
K or L, shifted 5. Code 114. Cursor left. 19
K or L, shifted 8. Code 115. Cursor right. 19
K or L, shifted 7. Code 112. Cursor up. 24
K or L, shifted 6. Code 113. Cursor down.

Report Codes

This table gives each report code, with a general description and a list of the statements and functions in which it can occur. In Chapter 21, under each statement or function, you will find a more detailed description of what the error reports mean.

Code	*Meaning*	*Situations*
Ø	Successful completion, or jump to line number bigger than any existing. A report with code Ø does not change the line number used by **CONT**.	Any
1	The control variable does not exist (has not been set up by a **FOR** statement), but there is an ordinary variable with the same name.	**NEXT**
2	An undefined variable has been used. For a simple variable this will happen if the variable is used before it has been assigned to in a **LET** statement. For a subscripted variable it will happen if the variable is used before it has been dimensioned in a **DIM** statement. For a control variable in a **FOR** statement and if there is no ordinary simple variable with the same name.	Any
3	Subscript out of range. If the subscript is out of range (negative, or bigger than 65535), error B will result.	Subscripted variables
4	Not enough room in memory. Note that the line number in the report (after the /) may not be complete on the screen,	**LET, INPUT, DIM, PRINT, LIST, PLOT, UNPLOT, FOR,**

Code	*Meaning*	*Situations*
	because of the shortage of memory: for instance, 4/2Ø may appear as 4/2. See Chapter 9.	**GOSUB**. Sometimes during function evaluation.
5	No more room on the screen. **CONT** will make room by clearing the screen.	**PRINT, LIST, PLOT, UNPLOT**
6	Arithmetic overflow: calculations have led to a number greater than about $1Ø^{38}$.	Any arithmetic
7	No corresponding **GOSUB** for a **RETURN** statement.	**RETURN**
8	You have attempted an **INPUT** command (not allowed).	**INPUT**
9	**STOP** statement executed. **CONT** will not try to reexecute the **STOP** statement.	**STOP**
A	Invalid argument to certain functions.	**SQR, LN, ASN, ACS**
B	Integer out of range. When an integer is required, the floating-point argument is rounded to the nearest integer. If this is outside a suitable range, error B results.	**RUN, RAND, POKE, DIM, GOTO, GOSUB, LIST, PAUSE, PLOT, UNPLOT, CHR$, PEEK, USR**
	For array access, see also Report 3.	Array access
C	The text of the (string) argument of **VAL** does not form a valid numerical expression.	**VAL**
D	(i) Program interrupted by **BREAK**.	At the end of any statement, or in **LOAD, SAVE, LPRINT, LLIST,** or **COPY**.
	(ii) The **INPUT** line starts with **STOP**.	**INPUT**
E	Not used	
F	The program name provided is the empty string.	**SAVE**

$10.95

330-011006

TIMEX Timex Computer Corporation Waterbury, Connecticut 06720